Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Sistema de Detecção e Prevenção de Fraudes de Clique para Redes de Anúncio

Paulo S. de Almeida

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientador
Prof. Dr. João José Costa Gondim

Brasília
2018

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Sistema de Detecção e Prevenção de Fraudes de Clique para Redes de Anúncio

Paulo S. de Almeida

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Prof. Dr. João José Costa Gondim (Orientador)
CIC/UnB

Prof. Dr. Marcos Fagundes Caetano Prof. Dr. Dino Macedo do Amaral
CIC/UnB

Prof. Dr. Ricardo Pezzuol Jacobi
Coordenador do Curso de Engenharia da Computação

Brasília, 22 de março de 2018

Dedicatória

Dedico esse trabalho aos grandes amigos que fiz ao longo dos anos, próximos ou não mais, e aos da minha família que me apoiaram por tanto tempo.

Agradecimentos

Agradeço aos professores e orientadores que me ajudaram a chegar na elaboração desse trabalho. Agradeço também aos vários sites que ajudaram nas pesquisas, como o *Google* e o *StackOverflow*.

Resumo

Este trabalho apresenta os conceitos e dinâmicas envolvidas nas práticas de fraudes de clique, e com base na literatura já existente e nas informações conhecidas da área propõe um sistema de detecção de prevenção desse tipo de fraude. O sistema toma como premisa o lado da rede de anúncio, um dos principais atores no ambiente de anúncios *online*. Para a validação do sistema, foram feitos 3 servidores, representando o publicador, a rede de anúncios com o sistema de defesa implementado e o anunciante, e um *bot* que tenta praticar a fraude nesse ambiente. Para trabalhos futuros, pensamos na adição de técnicas de computação mais refinadas ao sistema atual como *machine learning* ou *browser fingerprinting* para melhorar as funções já presentes no sistema, além da implemetação da proposta em um cenário do mundo real, com os devidos ajustes.

Palavras-chave: fraude de cliques, arquitetura de sistema, anúncios online

Abstract

This work presents concepts and dynamics involved in click fraud practices, and with basis on the already published literature and knowledge of the area proposes a system that detects and prevents this type of fraud. As a premise, the system is based and implemented on the ad network, one of the 3 main agents in the on-line ad environment. For the system's validation, 3 servers were made, representing the publisher, the ad network with the system implemented and the announcer, and a bot, that represents an user trying to commit click fraud. For future works, we thought about adding more refined computing techniques, such as machine learning or browser fingerprinting, to enhance the system's current functions, and applying the proposition to a real-world environment, making necessary changes.

Keywords: click fraud, system architecture, ads online

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A área de publicidade online cresce de forma contínua atualmente. O relatório da *eMarketer* [3] mostra um crescimento de quase 16% para os gastos nesse tipo de publicidade nos Estados Unidos, se comparado ao ano passado, totalizando um investimento de 83 bilhões de doláres. Essa quantia é mais de 40% dos totais investimentos em publicidade no país. De tal maneira, podemos ver que o comércio na área ganha foco, assim como formas de explorar seus aspectos técnicos com diferentes técnicas de ataque e de defesa.

Neste capítulo vamos introduzir as ideias básicas do sistema de defesa, brevemente introduzido na seção 1.2 e detalhado no capítulo 3. Falamos também sobre a motivação por trás desse estudo, e na seção de *outline*, mencionamos os capítulos presentes ao longo do texto e o que cada um deles irá abordar.

1.1 Motivação

A detecção de *click frauds* é um campo inerentemente nebuloso e que acarreta incertezas: o objetivo é, afinal, determinar a intenção por trás de um *click*, dadas apenas informações técnicas e gerais que são recebidas em todo tipo de *click* (nominalmente, requisições HTTP) e informações de navegação do usuário que executou o click. Mesmo pessoas experientes na área podem ter dificuldade em identificar a legitimidade de um *click* qualquer, já que essa conclusão geralmente depende do contexto. A análise leva em conta padrões de acesso normais e tenta localizar os padrões suspeitos. Procura então bater toda a massa de dados de tráfego do sistema contra o comportamento esperado de um usuário legítimo. Grande parte da subjetividade vem da determinação sobre o que seria esse comportamento legítimo, já que em geral não se tem como determinar o autor de um *click* fraudulento por meios além do seu *IP* (que pode ser facilmente alterado), e é difícil deduzir a ilegitimidade de fraudes complexas e planejadas que se aproveitam de brechas em técnicas comuns de detecção para essas atividades.

Veremos ainda que a literatura atual da área apresenta duas áreas faltosas: a análise de fraudes de clique na visão da rede de anúncios, com várias soluções focadas na visão do anunciante, ou até de uma junção de atores diferentes, como *ad networks* que também são publicadores, como Kitts et al. [1]; e, por motivos de ofuscação sobre o funcionamento desse tipo de aplicação, para prevenir atacantes de terem conhecimento sobre técnicas de defesa, ou até por questão de muitos desses estudos terem por trás deles *softwares* proprietários, não informam ou não entram em detalhes sobre suas implementações ou como chegaram a algumas opções de design.

1.2 Objetivo

Temos como objetivo nesse estudo propor um sistema de detecção de *click frauds* implementado pela *ad network* ou rede de anúncios, o agente mediador entre o publicador e o anunciante, que procura enviar ao usuário o anúncio que provavelmente terá maior chance de interagir com. A rede de anúncio também tem como interesse prevenir que *click frauds* acontecam em seu sistema; já que a rede é paga pelos anunciantes, e anunciantes querem apenas usuários legítimos. A premisa então é a seguinte: criar o sistema que impeça ou detecte ataques vindos de qualquer outro agente. O usuário e o publicador não são confiados, e apenas uma regra opcional no sistema conta com a ajuda do anunciante, que será elaborada sobre no capítulo 3.

Além disso, esse estudo tem como objetivo mostrar em detalhes como um sistema desses seria implementado, de maneira aberta a todos. A maioria da literatura atual, apesar de excelente, não explica à fundo como seus sistemas de detecção funcionam, e mostrar as questões e decisões envolvidas no design do nosso sistema, seus módulos e as interações entre eles.

1.3 Metodologia

Para estudar o assunto, assim como em várias outras áreas da segurança de dados e redes, é útil ver o problema de forma **dual**, isto é, procurar vê-lo sempre dos dois lados: ataque e defesa. O progresso técnico nessse assuntos muitas vezes acontece dessa maneira, com dois ou mais lados envolvidos e tentando constantemente estar à frente do outro, num jogo de ação e reação. Por exemplo, uma empresa tem dados sigilosos e deseja defender eles de usuários mal-intencionados. A empresa então cria um sistema que impede tais usuários de abusarem o sistema, talvez por meio de uma gestão de permissões. Para contornar isso, os usuários podem utilizar técnicas para enganar a gestão de permissão, burlando a chave única de algum usuário com permissões elevadas e garantindo acesso aos dados.

Em resposta a empresa implementa alguma forma de impedir que os inimigos roubem essa chave em primeiro lugar, e assim continua essa dança entre os dois lados.

Nesse olhar, apesar desse estudo propor por fim um sistema de defesa contra fraudes de clique, precisamos então elaborar o sistema mantendo em mente sempre os dois lados: como defender contra o atacante, e como atacar contra a defesa.

1.4 Outline

Ao longo desse documento vamos explorar o caminho tomado para a criação do sistema de detecção de fraude de cliques. No próximo capítulo veremos os conceitos básicos e os estudos já existentes na área para poder definer com clareza qual vai ser o objetivo do sistema desenvolvido. Em seguida, no capítulo 3, entramos em detalhes sobre a proposta do sistema, principalmente sua arquitetura e seu funcionamento. O capítulo 4 então fala sobre como essa proposta foi de fato criada, discutindo os aspectos de implementação. Os testes feitos para o sistema são descritos então no capítulo 5, juntos com seus resultados, e ao final, no capítulo de conclusão, discutimos os resultados do estudo e possíveis trabalhos futuros a serem adicionados ao atual.

Capítulo 2

Revisão Teória Básica

Para podermos identificar o sistema a ser desenvolvido de forma específica, precisamos primeiro definir os conceitos a serem utilizados ao longo do trabalho, e os problemas atuais já tratados pelas pesquisas atuais, assim nos levando a identificar quais aspectos não foram ainda propriamente estudados ou tratados. Esses seram os principais tópicos falados sobre neste capítulo.

2.1 Conceitos de Publicidade Online

Antes de especificar o que é o problema central que olharemos, *Click Fraud*, precisamos entender o cenário em que essa prática ocorre. No campo de publicidade online, temos 4 principais entidades a considerar:

- Anunciante (*Advertiser*), que quer levar o conhecimento de seu produto (podendo ser seu *site*, seu blog ou mesmo um produto físico que quer vender) a um publico alvo;
- Publicador (*Publisher*), que tem o seu proprio*site*e que entra em acordo com o anunciante, colocando o ad dele em seu*site*para que os visitantes do seu*site*o vejam e possam interagir com ele. O anunciante, em troca, paga o publicador, geralmente em relação a quantidade de interações positivas que o ad recebe;
- Usuário (*User*) ou Visitante, que entra no*site*do publicador e pode visualizar e clicar no anúncio, ou até efetuar uma compra ou outra ação financeiramente relevante para o anunciante;
- Rede de Anuncios (ou *Ad Network*), uma possível quarta entidade nesse modelo, que serve como um agente intermediário entre anunciantes e publicadores, podendo receber varios anúncios e atribuir alguns deles a vários publicadores diferentes.

O grande interesse nessa área de publicidade online vem pelo grande lucro que pode ser gerado nela. Para o cenário que involve uma *ad network*, o caso geral é que o anunciante procura a *network* para promover o que deseja, como seus produtos ou um fórum, equanto a rede, que já possui um contrato com vários *sites* publicadores, manda o anúncio para alguns deles. Para definir o quão bem-sucedida uma campanha de anúncio é, usamos os conceitos de **impressão** (*impression*), registrada quando qualquer visitante visualiza um *ad* e de **conversão ou ação** (*conversion* ou *action*), algo definido como objetivo do anúncio pelo anunciante, como a compra de produtos ou inscrição em um *site*. Os anunciantes e redes de anúncio procuram por bons números nesse conceitos: no caso geral, um publicador com um elevado número de impressões por ad deveria, naturalmente, levar a uma grande quantidade de *clicks* no ad, e isso por sua vez deveria levar vários visitantes a fazerem uma conversão, mas como diferentes tipos de usuários tem diferentes interesses, nem sempre essa escala realmente acontece. A rede cobra os anunciantes e paga os publicadores seguindo um modelo de interações desejadas. O modelo utilizado vária bastante e em geral é escolhido pelo anunciante, cada um contento vantagens e desvantagens. Os mais populares são:

- *Pay Per Impression* (PPI), no qual a alavanca de cobrança/pagamento é o número de visitantes que viram a publicidade. Em geral, impressões são relativamente fáceis de obter, e um número grande de impressões não necessariamente reflete em um número grande de conversões para o anunciante, então o pagamento por impressão é extremamente baixo.

- *Pay Per Mille* (PPM), uma variação mais realista do modelo PPI. Nesse modelo de pagamento, o publicador recebe o pagamento baseado em quantas mil impressões o ad gerou em seu *site*;

- *Optimized Pay Per Mille* (OPPM). O publicador é seletivo em sua escolha sobre para qual usuário mostrará o *ad*, fazendo sua decisão com base no perfil do usuário e dos assuntos que o anúncio trata. Como a impressão vinda de um visitante com mais interesse em potencial que um visitante aleatório é mais valiosa, esse modelo custa mais caro que o PPM. Sua implementação é mais complexa, porque precisa que o publicador mantenha perfis dos usuários com suas preferências, e o anunciante precisa colocar completa confiança no publicador para que ele de fato distribuia seu anúncio para usuários mais interessados.

- *Pay Per Click* (PPC), onde a base do pagamento é considerada o número de *clicks* que o ad recebeu no *site*. O preço de venda nesse caso é baseado principalmente no CTR (*Click-Through Rate*) do publicador, ou seja, a razão entre o número de usuários que acessou o *ad* e as impressões que*site*obteu pro mesmo, e;

- *Pay Per Conversion* ou *Pay Per Action* (PPA), em que os agentes envolvidos com a publicação são pagos por cada usuário transferido para o site anunciante que faz uma ação específica pré-definida pelo anunciante, como criar uma conta no site ou comprar um produto nele.

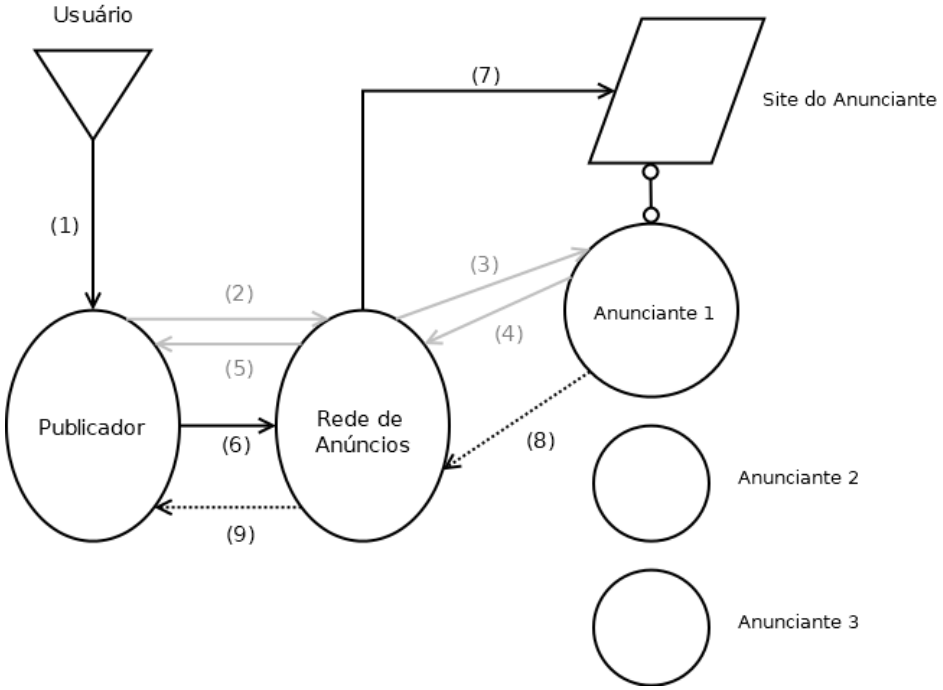


Figura 2.1: Possível cenário de publicidade online.

Na figura 2.1 temos uma possível implementação de um sistema PPC. O usuário visita o *site* publicador (1). O usuário então pede os dados de anúncio colocados no *site* publicador para rede de anúncios (2). A rede escolhe o anúncio que acredita que o usuário tem mais chance de se interessar em (3) (4) e envia seus dados para ele (5). Caso se interesse, o usuário então interage com o anúncio, sendo redirecionado para a página da rede de anúncios (6) e então para o site do anunciante (7). Terminado o processo, o anunciante escolhido paga a rede de anúncios pelo encaminhamento do usuário (8), que então paga também o publicador por transferir o usuário em primeiro lugar (9).

2.2 Fraude de Clique

Click Frauds são, basicamente, tentativas de fraudar o sistema de PPC em que um usuário ilegítimo, sem interesse real no que está sendo mostrado na publicidade, efetua um *click* na mesma. Existem vários possíveis motivos e interesses para a utilização de *Click Frauds*:

- Lucro própio, já que o *site* publicador ganha por *click*;
- Prejuízo financeiro, geralmente feito por concorrentes do anunciante, que, se bem sucedidos, fazem com que ele pague por *clicks* irrelevantes;
- Auto-promoção: Um sub-item do caso anterior. Quem comete a fraude gasta os recursos financeiros de concorrentes com *ads* na mesma área de pesquisa, e portanto consegue melhor promover seu próprio anúncio e adquire um tráfego humano e de maior qualidade que as empresas adversárias, e;
- Difamação, quando a fraude é feita com o objetivo de ser detectada e danificar a reputação do *site* que publica o anúncio, podendo até exclui-lo do contrato com o anunciante ou rede de anúncios.

O usuário ilegítimo pode ser um *bot* feito para automaticamente clicar no *ad* com certa frequência, um trabalhador contratado para clicar no *banner* várias vezes ao dia e até navegar o *site* publicado, ou até mesmo um usuário inocente que teve sua máquina comprometida, e então acessa o *site* do anunciante sem verdadeira intenção ou até conhecimento do que está acontecendo. As fraudes feitas automaticamente por *bots* ou programas são em geral mais fáceis de identificar, já que, por definição, operam por meio de algoritmos.

2.3 Protocolo HTTP

Para entender os aspectos técnicos envolvidos no desenvolvimento desse estudo, devemos nos referir ao protocólo que dá a base para que esse trabalho seja feito, o HTTP. A sigla significa *Hypertext Transfer Protocol* (ou Protocolo de Transferência de Hipertexto), e é a base de uma grande parte das funcionalidades da *internet*. A família de publicações RFC 7230 [4] [5] serve como uma fonte oficial do funcionamento e detalhes do protocolo. Em traduções livre, o HTTP é definido como "um protocolo a nível de aplicação sem estado para sistemas de informação hipertexto distribuidos e colaborativos."Em termos de funcionamento, cada envio HTTP é uma requisição (*request*), que em torno recebe uma resposta *response*. A resposta leva consigo um número que representa um estado, como 301 (endereço movido permanentemente) ou 200 (indica que a requisição foi respondida com sucesso). Apesar da maioria das interações envolvendo o protocólo serem baseadas somente na troca desses dois tipos de mensagens (*request* e *response*), outros métodos são usados para contornar limitações do dele, principalmente *cookies*.

Cookies foram feitos para permitir interações com estado por meio do HTTP. Assim, coisas como preferências de usuários em um *site* qualquer podem ser armazenadas e depois

verificadas pelo *site* quando o usuário voltar a visitá-lo. *Cookies* também permitem coletar informações do cliente e as receber de volta no servidor, técnica que será utilizada em uma das regras do sistema, descrita em detalhes no próximo capítulo.

2.4 Publicações Anteriores

Como já mencionado antes, a análise e o desenvolvimento de formas de *click fraud* pode ser dividida entre métodos de ataque e métodos de defesa. Vamos então rever o que a literatura atual tem a apresentar em questão dessas duas categorias:

2.4.1 Defesa

Muitos documentos que falam sobre a defesa contra *click frauds* focam em sistemas individuais e técnicas mais específicas utilizadas para o combate nesse sistema. Esses incluem os estudos de Kitts, B. et al. [6], em que falam sobre o sistema utilizado no *Microsoft adCenter* e as decisões de design e performance envolvidos em sua elaboração, e Xu, H. et al. [1], em que discutem em detalhe técnicas para serem usadas em um sistema de detecção no lado do anunciado.

Outros, porém, apresentam técnicas gerais de detecção ou prevenção, como Dawani, D et al. [12], no qual os autores elaboram e introduzem os vários conceitos pertinentes à discussão, falando tanto de técnicas de defesa quanto formas elaboradas de ataque.

Uma tática comun de defesa é obter o comportamento do usuário dentro dos *sites* do publicador e do anunciante. Dessa forma, pode-se analisar e comparar as ações do usuário com as esperadas de um usuário não-*bot*, como quanto tempo o usuário passa nas páginas do anunciante ou se sequer interage com ela. Como a definição do comportamento normal é subjetiva e depende da implementação, obstruir diretamente esse filtro involve também um estudo da parte do oponente.

2.4.2 Ataque

Ataques de *click fraud* podem ser divididos entre os automatizados e manuais. Os casos mais sofisticados de ataques manuais, em que trabalhadores são contratados para visualizar e interagir com os *ads* alvo, precisam de uma análise complexa para serem detectados, mas as tentativas mais básicas desse são em geral fáceis de detectar automaticamente, já que raramente usam técnicas de obfuscação para esconder o ataque. Por exemplo, um publicador clica nos ads no próprio *site* para obter lucro. Esse caso poderia ser detectado simplesmente verificando que os vários *clicks* do *ad* vem de um mesmo *IP*. O atacante pode responder utilizando um *IP* dinâmico ou *proxies* para contornar a defesa.

Botnets

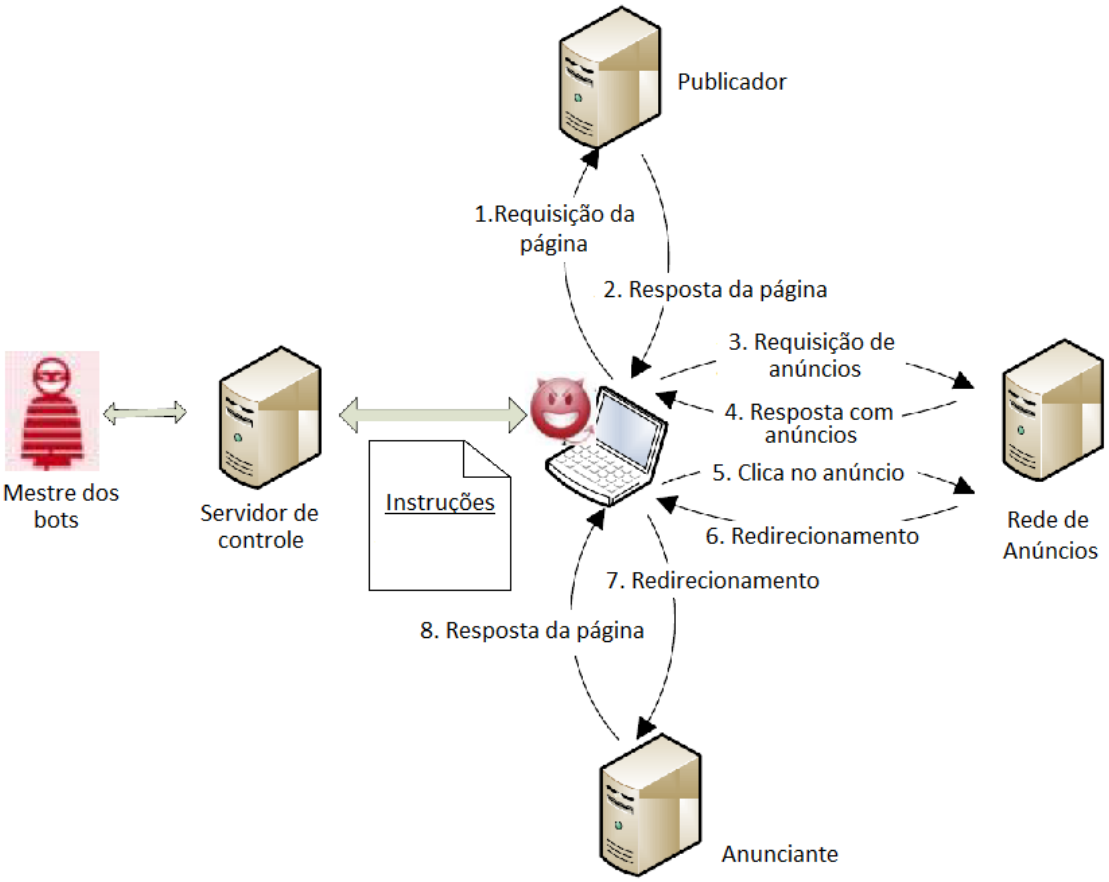


Figura 2.2: Exemplo de um ataque *botnet*. Adaptado de: [1].

Uma técnica comum e mais elaborada que as já mencionadas é o uso de *botnets*. Nesse tipo de ataque, o computador de várias vítimas é infectado com um *malware* especificamente feito para o ataque. Depois de comprometer várias máquinas, o malware entra em contato com um servidor (esse podendo ser único ou um sub-servidor, que também recebe ordems de um servidor mais central), e recebe anúncios para poder clicar, muitas vezes sem alarmar o verdadeiro dono do computador. Dessa forma, o ataque obtém várias caracteristicas desejáveis, pois tem acesso a vários *IPs* distintos e consegue se aproveitar do comportamento de um usuário humano para poder se disfarçar.

Esse método de ataque é complexo e difícil de detectar, e uma das principais formas de verificar seu acontecimento é uma análise da frequência de *clicks*. Como exemplos de documentos já publicados sobre casos desse tipo, temos o estudo do caso de um esquema de *click fraud* nomeado "*ClickBot A*" [7]. Nele vemos os detalhes do funcionamento dessa *botnet* e seu impacto. Outras referências similares, em que são analisados distintos casos reais de ataques via *click frauds*, podem ser vistos em [8], [9] e [10].

Ataques de Baixa Frequência

Um outro tipo *click fraud* é o chamado Ataque de Baixa Frequência. Esse método de fraude se aproveita de uma técnica de defesa em que a frequência de *clicks* é analisada. Como os anúncios tem uma quantidade esperada de *clicks* em um período (e essa quantidade esperada pode ser feita ainda mais precisa com a maturidade do sistema, em particular com uma maior base de dados para fazer análises), o sistema detecta e fica alerta quando um anúncio tem um CTR muito maior que o esperado. As tentativas de fraude mais básicas em geral focam em quantidade de tráfego que conseguem atingir para o *ad* alvo (quanto maior a quantidade de *clicks*, maior seu ganho), e podem ser detectadas com essa defesa. Além disso, esse tipo de ataque muitas vezes é aplicado por *botnets*, dificultando ainda mais sua detecção.

Os ataques de baixa frequência contornam essa suposição ao não deixar os anúncios alvo serem clicados mais que uma certa quantidade de vezes por dia ou outro período de tempo. Como o ataque não chama atenção pela sua frequência, ele é em geral mais difícil de detectar e foca em ser executado por longos períodos de tempo para que o atacante consiga algum retorno monetário significativo. As máquinas comprometidas se comunicam com um servidor para saberem se podem clicar no *ad*, e o servidor pode verificar a quantidade de vezes que já ordenou a interação com o *ad* no dia. Caso a quantidade seja menor que a frequência máxima estabelecida (por exemplo, 20 *clicks* por dia para um dado domínio), então o servidor responde à maquina que pode executar o click. É possível perceber que se tratam de fraudes elaboradas, e cuja detecção leva meses de trabalho de times especialistas no assunto, como visto em trabalhos já mencionados ([7] e [8]).

Capítulo 3

Proposta

Antes de podermos testar ou mesmo implementar o sistema que desejamos, precisamos primeiro criar uma proposta precisa do que será feito, aprofundando a mesma com propostas de arquitetura e funcionamento geral. Com isso em mente, neste capítulo veremos esses aspectos da criação do sistema, principalmente a arquitetura dele e as regras, parte vital do seu funcionamento. A arquitetura aqui levantada foi elaborada com base na proposta de Kitts et. al [6]. As regras **JavascriptEnabledRule** e **ExternalBehaviorRule** tiveram como base o trabalho de Xu et. al [1], enquanto as outras regras (**BlacklistRule**, **HumanTimerRule**, **PagesLoadedRule**, **AcceptLangRule**, **TimePeriodRule**, **UserAgentRule**, **DoNotTrackRule** e **RedirectTimeRule**), além das categorias que classificam as regras (*offline* ou *online*, indicativa ou decisiva), são nossas contribuições ao estudo.

3.1 Descrição Geral

O sistema proposto nesse estudo visa atender a necessidades genéricas de agentes com o papel de *Ad Network*, ou seja, uma entidade que obtém anúncios dos *advertisers* e distribui esses anúncios para os *publishers*. A premisa é prevenir e detectar fraudes da parte dos *publishers* e/ou *users*, contando com o auxílio dos *advertisers*. Parte do processo de decisão sobre a validez de um *click* involve os dados de comportamento do usuário, que seriam providos pelo *advertiser* depois que o usuário que clicou no anúncio já tiver interagido com o seu *site*. Apesar de muito útil para auxiliar na detecção de usuários suspeitos, essa parte da detecção pode ser pulada caso não se queira ter a confiança no anunciante como um premissa.

3.2 Arquitetura

A arquitetura geral do sistema se baseia no sistema elaborado pela Microsoft [6], porém simplificado e sem preocupações sobre escalabilidade ou performance. A idéia é registrar toda impressão e *click* gerado para cada anúncio gerenciado pela *network*. O sistema guarda todos os detalhes possíveis sobre os *HTTP requests* que recebe, englobando as dois tipos de interação mencionados. Dessa forma se tem uma base de dados sólida e sempre útil para a detecção de comportamento suspeito em vários aspectos (por exemplo, uma quantidade grande de *clicks* em um *site* publicador específico, ou em anúncios de um anunciante específico). Esse tipo de persistência permite também o *replay* de correntes de ações antes executadas no sistema, caso necessário.

Na figura 3.1 a seguir mostra uma representação da arquitetura do sistema geral e seus módulos, cujos funcionamentos básicos serão elaborados a seguir.

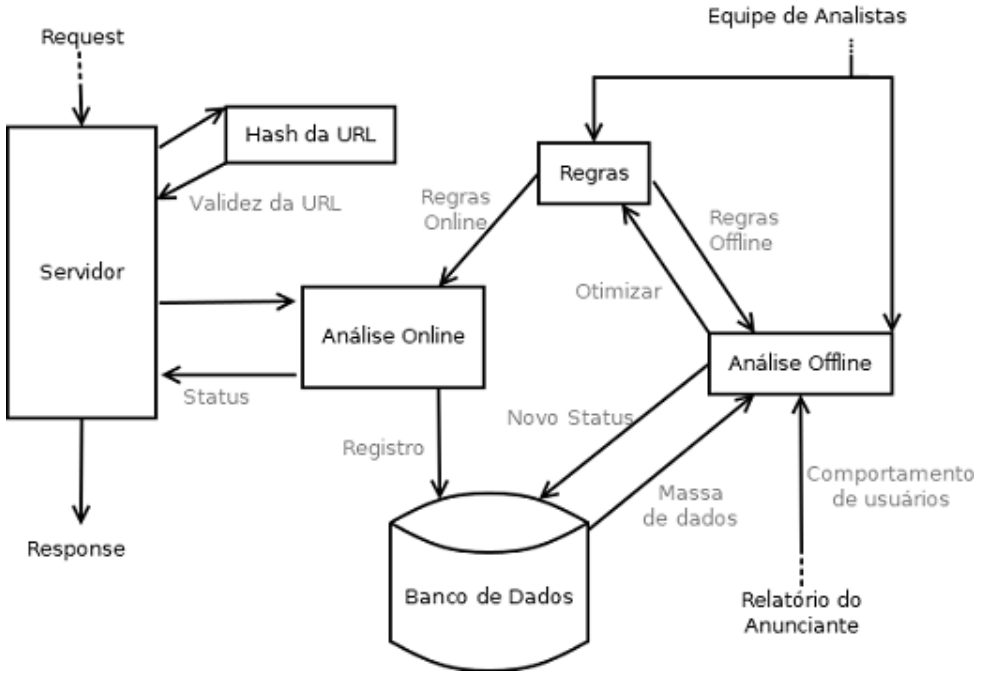


Figura 3.1: Esquemático da arquitetura básica do sistema.

O funcionamento geral do programa se dá da seguinte maneira: Um *request* chega ao servidor. A *URL* que esse *request* pediu é analisada, e passa pelo módulo ***Hash* da *URL*** caso seja uma *URL* de anúncio. Tomadas as medidas necessárias, e caso o endereço seja decidido como válido, passamos para o sistema de detecção em si. As informações recebidas no *HTTP request* são enviadas para a **Análise Online**. Esse módulo consulta a combinação de regras online atuais, que vêm do módulo **Regras**. Feita a análise desse módulo, as informações do *request* estudado, junto com o *status* sobre a validade do *click*,

são armazenados no **Banco de Dados**. O *status* é também retornado para o servidor. Por fim, o servidor retorna a página apropriada ao pedido feito pelo cliente, independente do estado observado pela **Análise *Online***.

Em paralelo a esse procedimento atua o resto do sistema, nominalmente a Análise *Offline*. A Análise *Offline* faz seus procedimentos em períodos pré-determinados de tempo, que podem ser mudados de acordo com o que a equipe de monitoramento achar necessário. O ponto de entrada para essa parte da arquitetura é o envio dos dados já coletados no Banco de Dados para a Análise *Offline*. Essa então utiliza o conjunto de regras *offline* determinados anteriormente no módulo Regras, e passa os dados pelos diversos testes. Para ajudar a chegar na conclusão, o módulo pode também usar um relatório de comportamento de usuários, enviado pelo próprio anunciante, caso esse o provenha. Determinado o estado do *click*, o módulo por fim envia o estado de validez do *click* atualizado de volta para o banco.

Ainda no módulo *offline* pode ocorrer o treinamento de regras, em que conjuntos diferentes de regras podem ser usados e comparados em performance para determinar o estado de *clicks* já armazenados. Esse processo seria gerenciado pela equipe de analistas, que determinaria os conjuntos de regra a serem testados e os com estado já considerado "resolvido"que seriam usados para teste.

Existe ainda a arquitetura de navegamento das páginas do *site*, que é feita para coletar as informações necessárias para realizar alguns dos testes. Depois de acessar o *site* via *ad* no domínio do publicador, o visitante vai para o site da *ad network*. Essa primeira página redireciona o usuário para a segunda página no domínio, e envia para o usuário um *cookie*, além de possuir uma imagem de 1 pixel. A segunda página no processo então confere a presença do *cookie* no *request* para ela, além de possuir um *link* escondido que volta para a primeira página. Por fim, a segunda página redireciona o usuário para o seu destino final, o *site* do anunciante.

Regras

O principal método de detecção de fraude usado no sistema são as Regras. A ideia é testar todo *click* recebido contra um conjunto de regras, e assimilar os resultados desses testes e assim tomar uma decisão sobre a intenção do *click*. As regras podem ser classificadas em relação ao sistema em que são usadas, podendo ser **Offline** ou **Online**:

- ***Online***: As regras *online* devem ser rápidas e de baixo impacto para o sistema como um todo, já que são executadas em tempo real, e idealmente não devem impactar a performance do serviço provido pela *ad network*. Elas são executadas a cada *http request* identificado como um *click* recebido por um *ad*. Para *sites* com grande tráfego, isso pode levar a dezenas ou até centenas de execuções por segundo. Essas

regras então são utilizadas para cada *click* e devem depender minimamente de partes custosas do sistema, como acesso ao banco de dados de *clicks*.

- ***Offline***: Em contraste, esse tipo de regra se preocupa somente com *clicks* que já passaram pelas regras *online* e que já estão presentes na camada de persistência. Regras *offline* conseguem fazer uma análise detalhada e profunda dos *clicks* e procuram principalmente por padrões suspeitos de acesso ao site. É o caso para *clicks* constantes em certos períodos de tempo ou comportamento de pedidos de *sites* anormais comparados ao de um usuário comum, e/ou de um navegador.

Além dessa categoria, dividimos as regras em relação à sua importância na hora da análise de validade, também entre duas categorias, **Decisiva** ou **Indicativa**:

- ***Decisiva***: Caso o *click* analisado falhe no teste de uma regra decisiva, o click será automaticamente considerado não válido. Essas regras representam algo que o sistema considera essencial para um click legítimo, como um campo *"User-Agent"* vazio ou anormal no *request* do *click* recebido.

- ***Indicativa***: As regras indicativas apenas retornam indícios de fraude. Dessa forma, se um *click* não passar em um desses testes, não será necessariamente considerado frauduloso ou suspeito. Em termos de implementação essas regras tem pesos individuais, determinados antes da execução e mudados quando necessário. Dessa forma, diferentes conjuntos de regras podem ser usados para teste e a efetividade de diferentes pesos e combinações pode ser verificada.

Vamos então listar as regras decisivas e como funcionam, além das razões por trás de seus *designs*.

BlacklistRule

A primeira etapa do sistema de detecção de fraudes é a análise do IP identificado no HTTP Request. Essa análise inclui:

- Trivialmente, se o *IP* registrado do próprio publicador que enviou o usuário ao *site* anunciate, e se o endereço IP é de fato um endereço válido;

- A comparação com os *IPs* já guardados em uma *blacklist* de *IPs* já documentados como praticantes de *click fraud*. A lista é construida a partir de IPs que o sistema já identificou como praticante de fraude várias vezes, e;

- A geolocalização do *IP* é armazenada. Esse critério não desvalida o *click* automaticamente, mas caso o endereço seja de uma localização comunmente usada pra

práticas de fraude já colocam o sistema em estado de suspeita sobre o *click*. Dependendo da quantidade de fraudes detectadas da região, o bloqueio do *click* com base nesse critério é uma opção.

Caso o *click* seja bloqueado já nessa parte, será registrado no sistema que o publicador direcionou um *click* provavelmente malicioso. O sistema faz então uma análise do publicador, e decide se é um agente fraudador ou não, com base no percentual de *clicks* maliciosos identificados como vindo dele.

HumanTimerRule

A base dessa regra é que existe um tempo mínimo de reação entre um a visualização do anúncio online e a intereção do usuário com o mesmo, definido pelo tempo mínimo de processamento visual de uma pessoa. Enquanto o estudo feito por Thorpe, Simon et al. [11] aponta um valor de 0.15 segundos, outras pesquisas na área como a da *Census at School Canada* [12] e principalmente a da *Human Benchmark* [2] nos dão uma base para o tempo mínimo de reação entre a visualização e a reação das pessoas. Portanto, tomamos o valor a ser utilizado como 0.2 segundos. Então, qualquer interação vinda de um mesmo usuário com período menor que os tempos de reação mais rápidos obtidos é, com certeza, não-humano.

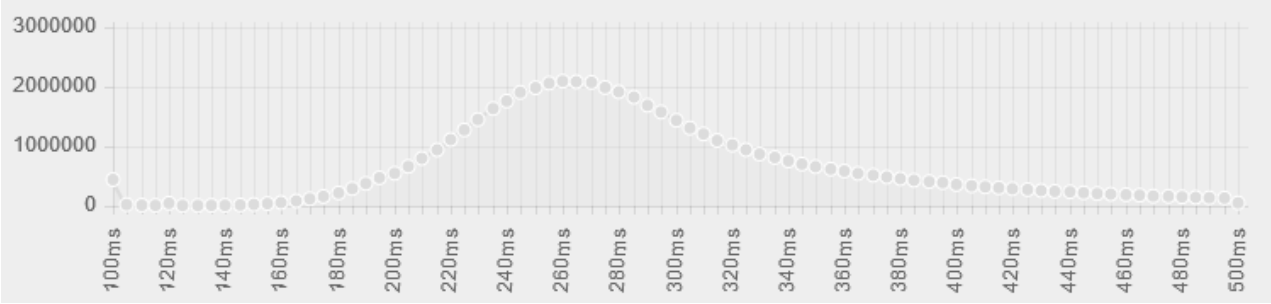


Figura 3.2: Número de acessos por tempo de reação. Fonte: [2].

Esse valor não leva em consideração que um usuário legítimo demorará um tempo entre carregar o *ad*, se interessar na campanha e interagir com ela. Com isso em mente, aumentamos o tempo aceito pela regra para 0.5 segundos. Dessa forma, podemos até identificar tráfego inválido por humanos contratados para *click fraud*, já que não levariam tempo pensando se estão interessados no anúncio fornecido e só vão clicar em anúncios o mais rápido possível.

PagesLoadedRule

Essa regra **offline** precisa do histórico de navegação do usuário para determinar se ele realmente se assemelha a um usuário normal. Para isso, procura por páginas que um usuário que interage com o *ad* deveria ter visitado, como a imagem do próprio *ad* depois de carregá-lo, e também por páginas que não deveriam acontecer, como um link para uma imagem escondido na segunda página de redirecionamento. Usuários legítimos nunca carregariam essa página, então o acesso a ela leva à conclusão que se trata de um atacante.

AcceptLangRule

Um conceito parecido com o da regra *DNTRule*, essa regra verifica se um *header* específico do *HTTP request* feito pelo usuário. Esperamos que usuários legítimos estejam usando navegadores *web*, que em geral tem esse campo setado da mesma forma para toda a requisição feita pelo usuário via navegador. A regra verifica se o usuário envio um campo *AcceptLang* preenchido e válido, ou comumente utilizado por *navegadores*. Em contraste com a regra *DNTRule*, esta é uma regra decisiva; falhar nesse teste indica um usuário que não está utilizando um *browser* convencional, e portanto muito provavelmente não é legítimo.

E agora, as regras indicativas:

JavascriptEnabledRule

Essa regra foca na detecção de *Javascript*. No estudo de Xu, H. et al [1], é levantada a estatística que pelo menos 98% dos usuários da *internet* tenham *Javascript* habilitado. A aceitação de *Javascript* é então usada como um critério para identifcar *click fraud*. Decidimos também adotar essa métrica. Detectamos a habilitação do *Javscript* utilizando um *cookie*, que será setado no primeiro endereço dentro do domínio da *ad network* que o usuário acessa. A presença desse cookie na página seguinte, também no domínio da rede de anúncios, é usada como indicativo que o usuário tem os dois habilitados, e portanto passa no teste.

Os dados de Priebe, J. em 2009 [13] apontam que 3.7% dos usuários bloqueiam *cookies* e uma quantidade insignificante bloqueia *Javascript*, enquanto o estudo mais recente de Winnicki, A. em 2016 [14] dá as percentagens de 99.93% para o uso de *Javascript* e 98% para os *cookies*. Assim, podemos assumir que essa parte do programa tem uma pouca possibilidade de levar a falsos-positivos.

TimePeriodRule

Essa regra **offline** procura por padrões de acesso de tempo pelo mesmo usuário dentro de um período longo de tempo. Acessos constantes e rápidos são um característica não esperada para usuários normais. Para podermos identificar esse tipo de acesso então, utilizamos 2 critérios: No primeiro, classificamos todo usuário que tenha 3 ou mais acessos em menos de período curto de tempo (proposto como 30 segundos no projeto, mas podendo ser alterado) como fraudador. Já no segundo, procuramos por 5 ou mais acessos dentro de um período mais longo de tempo (proposto como 10 minutos, mas novamente, podendo ser alterado) em que o tempo entre esses acessos é constante ou próximo a tal, outro padrão de acesso suspeito.

UserAgentRule

Verifica se o *request* tem um campo *"User-Agent"* válido. Essencialmente, *requests* que não tenham esse campo preenchido com o valor esperado de um *browser* (por exemplo, caso esteja vazio ou que indique ser feito em alguma linguagem de programação), o *click* não passará nesse teste. Apesar de ser uma boa maneira de detectar usuários suspeitos, da parte dos atacantes é trivial mudar o valor desse campo em suas requisições *HTTP*, então não é um método sem falha.

DoNotTrackRule

DoNotTrack (ou *DNT*) é um campo que geralmente é reservado para browsers no modo privado. Ele indica para o site que não deve manter informações sobre o visitante. Como não é algo que um atacante se preocupa com geralmente, já que estaria usando *proxies* para agir em primeiro lugar, e que muitas vezes não é setado, usamos esse parâmetro para julgar a legitimidade do *click*, de forma que, ao contrário das outras regras até agora, passar nesse teste aumenta levemente o valor agregado nas regras indicativas, e falhar nele não altera o mesmo.

ExternalBehaviorRule

Uma parte opcional para o auxílio na detecção de fraudes são logs de comportamento de usuário, providos pelo anunciante. Anunciantes tem incentivo para dividir com a rede de anúncio os históricos de navegação de usuários que considerar suspeitos, já que eles em geral tem mais a perder com comportamento fraudulento. Por exemplo, uma conversão não-monetária inválida não demonstra real interesse no produto anunciado, efetivamente significando que parte do dinheiro gasto pela campanha não trouxe verdadeiro retorno de investimento. O formato desses *logs*, baseados nos que foram definidos por X. Hu et al

[1], é mostrado na tabela 3.1. Além disso, essa regra é **offline**, já que depende das ações do usuário depois de já ter passado pelo sistema.

Categoria	**Campos**
Clicks	Na página inicial
	Nas outras páginas
Mouse Scrolls	Na página inicial
	Nas outras páginas
Mouse Events	Na página inicial
	Nas outras páginas
Tempo passado	Na página inicial
	Nas outras páginas
Número de páginas visitadas	Total

Tabela 3.1: Relatório de comportamento do usuário no *site* anunciante

Essas informações ajudam a diferenciar usuários fraudadores de usuários legítimos. *Bots* mais simples podem ser instruidos para apenas requisitar os *links* dos anúncios na página, e assim não tem um mouse propriamente implementado, ou não simulam propriamente a quantidade de mouse events e scrolls que um usuário humano legítimo geraria. Apesar disso, temos que considerar que alguns dos usuários que acessam os *ads* podem estar em plataformas mobile, que não possuem um *mouse* e não podem gatilhar Mouse Events ou Mouse Scrolls, portanto esses campos são considerados irrelevantes na análise caso sejam. Os campos de tempo passado no *site* principalmente ajudam a identificar usuários humanos ilegítimos: uma pessoa realmente interessada no conteúdo da página vai olhar o que está escrito, navegar as opções e levar tempo pensando se vai continuar navegando ou se vai tomar uma ação, como comprar um produto. Já o usuário fraudador, que pode conseguir ter mouse events e *clicks* comparáveis com os de um usuário interessado, só está interessado em clicar na maior quantidade de ads que conseguir, e vai ficar muito menos tempo nas páginas do *site*.

Existem também usuários ilegítimos que são instruidos a fingir o comportamento de um usuário interessado: eles podem mover o mouse e o scroll algumas vezes na página inicial para enganar detectores de fraudes mais simples. Para combater isso, o *log* do sistema deve registrar também as ações do usuário nas outras páginas do *site*. A falta de comportamentos considerados normais em usuários legítimos nessas outras páginas, como a falta de *clicks* ou passar poucos segundos nas outras páginas, pode indicar uma fraude de click. É importante notar, porém, que esse tipo de análise pode se aplicar também a visitantes legítimos, mas que ficaram desinteressados no conteúdo do anunciante. A análise da fraude a partir dessas informações deve ser mais cuidadosa para evitar o bloqueio de um usuário não-malicioso sem necessidade, mas para efeitos de cobrança ao anunciante,

a interação de um usuário desinteressado no produto equivale a um *click* inválido, já que nenhum dos dois vai gerar uma conversão.

RedirectTimeRule

Ao testar outras regras, notamos que o tempo redirecionamento mínimo entre uma página e outra é muito mais rápido em um *browser* do que para o *bot*. Enquanto o navegador responde a um campo "*meta refresh*"de uma página com tempo de redirecionamento igual a 0 em aproximadamente 0.5 segundos, o *bot* elaborado para o teste do sistema não conseguiu superar a marca de 1 segundo, com uma média mais próxima de 1.1 segundos. Com isso chegamos a essa regra, que verifica se o tempo de pedido entre a primeira e a segunda página da rede de anúncios é maior que o esperado. Apesar de ser uma boa maneira de identificar um usuário *bot*, não deve ser implementada como uma regra decisiva, já que o tempo de redirecionamento em uma página com "*meta refresh*"depende também da agilidade do *browser* que o usuário está usando, e alguém com um computador mais antigo poderá acabar falhando no teste. Mesmo assim, recomendamos um peso relativamente alto para essa regra.

Capítulo 4

Implementação

Nessa sessão do estudo, vamos aprofundar nas decisões de arquitetura de sistema, e como ele foi implementado e então testado. Em particular, serão discutidos os sistemas de defesa e ataque criados, e as implementações por trás de seus desenvolvimentos. Esses detalhes de implementação para ambos os sistemas são nossa contribuição.

Todo o projeto foi desenvolvido na linguagem de programação *Python* 3.6, e os dados de persistência foram armazenados em um banco de dados PostgreSQL, versão 9.6.

4.1 Sistema de Defesa

No começo do processo de um *click* legítimo, o usuário visita o *site* do publicador. O sistema de prevenção já começa a agir nessa etapa, pois a página *HTML* que o servidor do publicador envia ao cliente como *HTTP Response* deve conter links de redirecionamento ao servidor da *ad network*. O browser do cliente (ou o *bot* no caso de uma tentativa de fraude) segue esses links, enviando um novo *HTTP Request* para as páginas escolhidas. O *request* enviado nessa parte pode ser do tipo CORS (*Cross-origin resource sharing*), um *request* específico para recebimento de recursos vindos de um outro domínion (no caso, os *ads* que vem da rede de anúncios), mas em geral esse tipo especial de request não é necessário quando o *request method* for do tipo *GET*. O servidor da *ad network* então decide qual *ad* o usuário vai receber com base nos dados do usuário, como o seu endereço *IP*, preferências de linguagem, *site* de referência, entre outros, para determinar qual dos anúncios na rede é o mais provável de interessar o visitante. A *URL* enviada como link deve passar por uma encriptação antes de ser enviada para o usuário. Essa encriptação recebe como parâmetros os dados recebidos no *HTTP Request* feito pelo usuário para a rede, como seu IP, qual o*site*de referência, entre outros, e cria um hash com essa base e uma chave privada. Dessa forma, a *URL* criada será única, e caso o usuário acabe clicando no *ad*, o servidor poderá decriptar a *URL* recebida e confirmar se de fato o anúncio clicado

foi enviado para tal endereço IP em tal *site*. Confirmado que o anúncio é legítimo, o *click* do usuário pode então passar a ser analisado pelo resto do sistema. A figura 4.1 ilustra a etapa de criação e envio da *URL*.

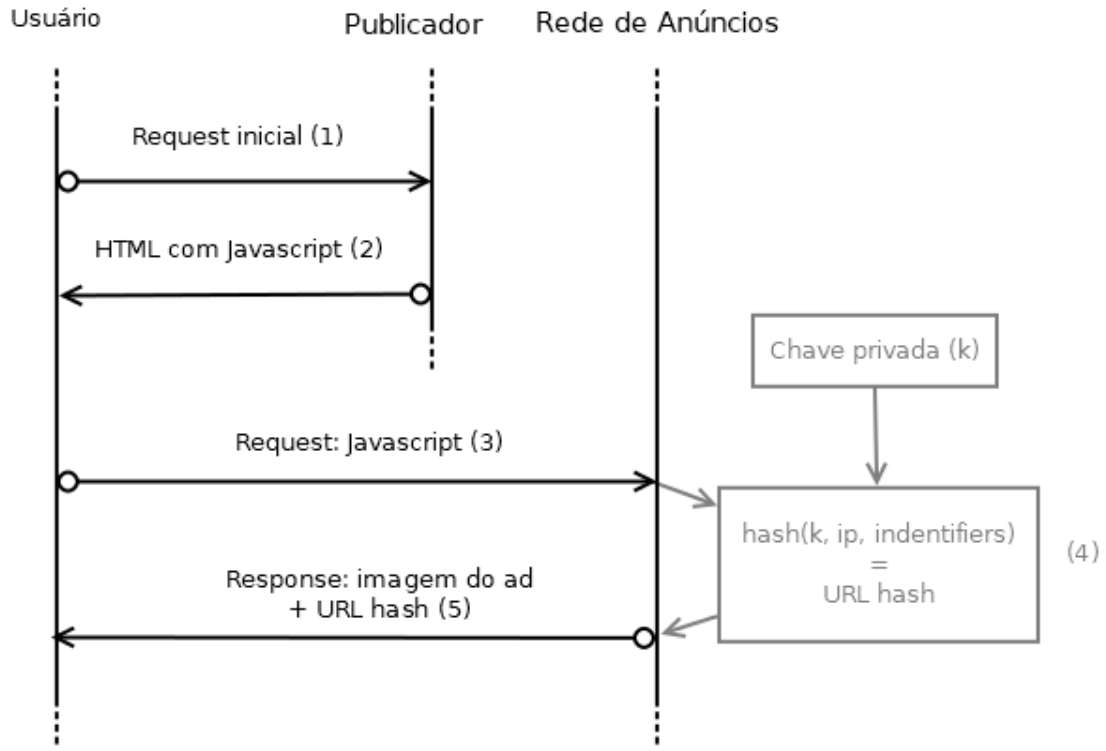


Figura 4.1: O processo do envio da *URL* de *ad* com identificador.

Em termos do funcionamento da captura e análise dos *clicks*, o sistema, como antes mencionado, redireciona o usuário 2 vezes: 1 vez para outra página do *site* da rede de anúncios, e dessa página para o destino do usuário, que seria o *site* do anunciante. Além de necessário para a aplicação de algumas das regras, cada uma das páginas também utiliza certas regras nas visitas que recebe.

- Na primeira página, **adRequest**, são aplicadas as regras **HumanTimer**, **Blacklist**, **AcceptLang** e **DNT**. A regra **HumanTimer** é aplicada aqui já que julga o tempo entre o carregamento do *ad* e o acesso do usuário ao seu *link*. As demais regras são escolhidas por já poderem ser testadas logo na primeira visita, e;
- Na segunda página, **redirect**, são feitos os testes de **UserAgent**, **Javascript** e **RedirectTime**. Esses testes necessitam ou funcionam de maneira mais confiável ao serem feitos no segundo acesso do usuário.

Para conectar um acesso ao outro do mesmo usuário, já que são para caminhos diferentes, capturamos todos os acessos à página **adRequest** em uma lista. No primeiro

momento, esse *request* não é registrado no banco, apesar de já ser julgado pelas primeiras regras. Quando entra uma requisição para a página **redirect**, os dados dela são comparados com os das requisições já na lista do **adRequest**. São comparados os campos que esperamos que sejam os mesmo, como *IP*, *user_agent* e *accept_lan/g*. Depois dessa comparação, vemos se a diferença entre os tempos de acesso é menor ou igual à esperada, que definimos como 3 segundos. Se sim, consideramos esses 2 acessos como sendo do mesmo usuário, e então fazemos o resto da análise com eles.

Mesmo que o *click* seja suspeito ou já determinado como sendo falso ao longo de todo o processo, o sistema não bloqueia o acesso do visitante a qualquer recurso ou parte que um usuário normal teria. Existem dois principais motivos para isso:

- Resiliência a ataques de tentativa e erro. Se o sistema bloqueasse todo *click* e usuário suspeito de interagir com o *site*, atacantes poderiam usar esse bloqueio como uma indicação que o sistema reconheceu algo suspeito no seu esquema de fraude, e assim iriam colher mais detalhes sobre o funcionamento do sistema e eventualmente criar esquemas que contornam as especificações de defesa.
- Falsos positivos. O sistema pode acabar concluindo errôneamente que um usuário legítimo é um usuário malicioso. Bloquear o usuário nesse caso é prejudicial para o *site* anunciante, que vai perder um possível interessado em seu produto.

A rede de anúncios pode esconder seu funcionamento interno ao não informar diretamente ao anunciante quais os *clicks* considerados fraudes e por quais motivos foram bloqueados. Esse método é útil para prevenir que anunciantes ou até mesmo publicadores maliciosos (o último podendo se passar por anunciante legítimo) testem o sistema por tentativa e erro, colocando um anúncio na rede e então atacando ele com configurações diferentes de *click* bots para deduzir quais os critérios usados para determinar quais são *clicks* falsos. Para não deixar o anunciante sem nenhuma indicação do desempenho da detecção de fraudes, sistemas de rede de anúncios, como o *adCenter* da *Microsoft* [6], geralmente informam ao usuário a quantidade ou percentual de *click*s que foram bloqueados.

4.2 *Bot* de Ataque

Para poder testar o funcionamento do sistema, criamos também um *bot* de cliques automático. O design do bot é focado em conseguir o maior número de *clicks* possíveis por segundo, sem ser registrada fraude por parte do sistema de defesa. Com esse objetivo em mente, o *bot* desenvolvido tem então vários níveis de funcionalidades para imitar o progresso de um atacante em entender como a defesa funciona e reage ao seu ataque.

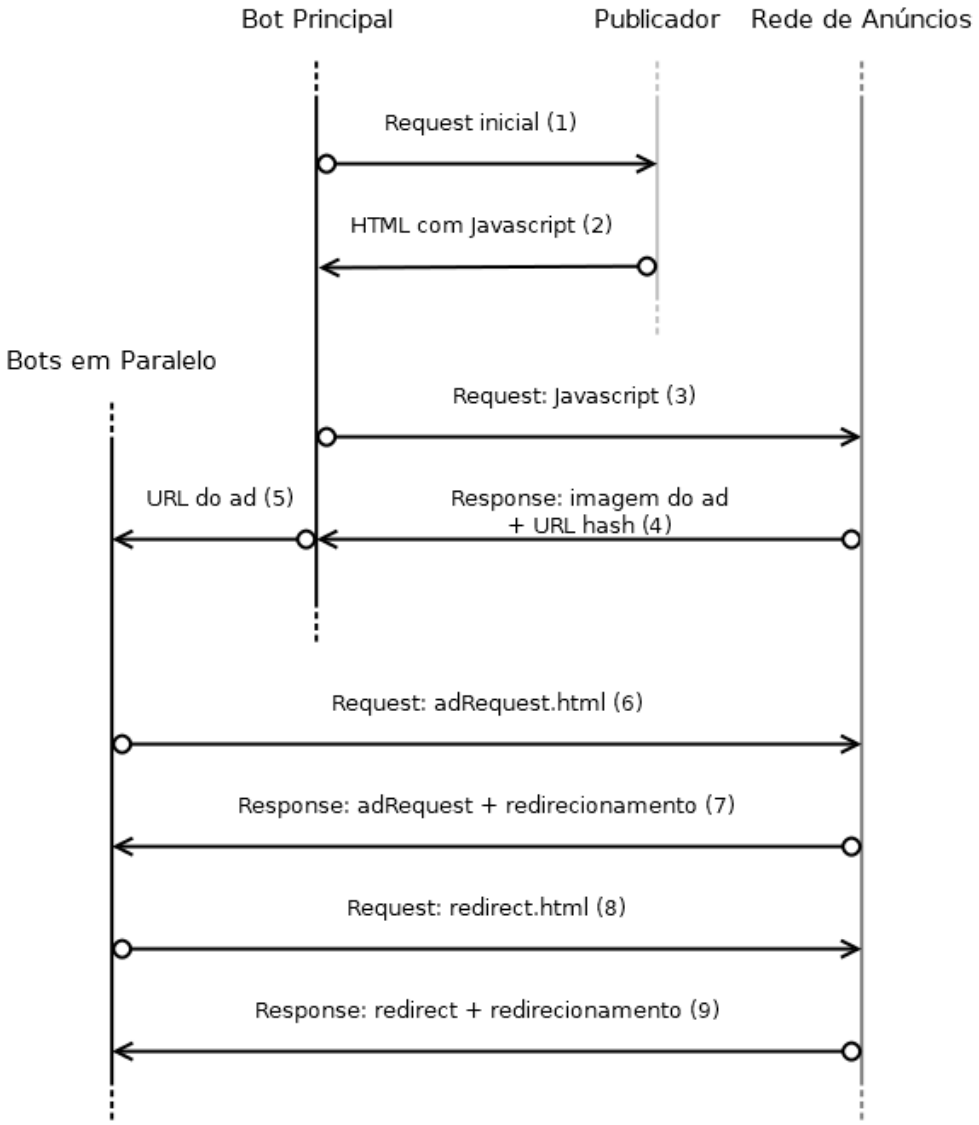


Figura 4.2: Esqueleto de funcionamento básico do *bot*.

O *bot*, assim como o sistema, foi desenvolvido em *Python* 3.6, e conta principalmente com a biblioteca *"requests"* do *Python* para fazer o papel de cliente para os servidores dos envolvidos no cenário. Primeiramente o bot é simplístico, e um esquema mostra seu funcionamento básico na figura 4.2. Antes da execução, o *link* do *ad* é capturado, e o programa recebe essa *URL* como entrada. Assim, o programa simplesmente acessa o *link* várias vezes por segundo, utilizando execução concorrente para fazer vários *requests* rapidamente. O *bot* então tem vários módulos para que todas as habilidades do sistema desenvolvido sejam testadas individualmente e em conjunto.

- **NormalAccessModule**: Acesso padrão aos sites. O programa começa a acessar os sites na ordem esperada de funcionamento para obter sempre *links* de *ads* legítimos.

Esse módulo é feito com um conhecimento do sistema em si em mente, já que o *link* oferecido do *ad* poderia ser utilizado para quaisquer usuários sem a proteção do *hash* da *URL* do mesmo. Esse módulo é um dos primeiros a ser implementado, já que o envio da *URL* do *bot* principal para os *bots* em paralelo necessita dele para funcionar corretamente.

- **HumanTimerModule**: O programa é aprimorado para fazer os requests em um espaço razoável de tempo, de modo que não será possível deduzir a validade do *click* só pelo tempo de acesso entre cada *request* feito.
- **HeadersModule**: Legitimação dos *HTTP headers* a serem usados no acesso. Essa regra inclui *headers* pré-definidos nos *requests* enviados pelo *bot*.
- **DNTModule**: Esse módulo inclui o campo *DNT* com valor 1 aos *headers* enviados. Esse módulo é separado do **HeadersModule** já que o campo DNT é completamente opcional, e sua inclusão como fator determinante da vericidade do *click* não é intuitiva como o uso de outros *headers* com valores usuais.
- **WideAccessModule**: Acessa todos os links passados. O bot acessa a todos os *links* enviados para ele pela *response* inicial vinda do *ad*. Essa é outra medida feita com o conhecimento do funcionamento da defesa, que verifica os *links* acessados pelo cliente ao longo do processo de redirecionamento.
- **SelectiveAccessModule**: Uma extensão do **WideAccessModule** que seleciona quais endereços deve carregar para simular um navegador verdadeiro. Por exemplo, com esse módulo, imagems escondidas nas páginas não seriam carregadas.
- **RandomTimeModule**: Mudança no padrão de acesso. O atacante começa a adicionar um gerador de números aleatórios para quebrar os padrões de acesso do *bot* (por exemplo, em vez de 4 acessos com intervalo de 5 segundos entre cada, os intervalos poderiam ser de 4 segundos, 1 minuto, 30 segundos e 10 segundos.)
- **CookieModule**: Esse módulo se encarrega de guardar e utilizar propriamente todo *cookie* enviado pelas páginas visitadas no processo do *click*.
- **CompleteAccessModule**: Manda *requests* para todas as páginas que seriam esperadas de um navegador normal, além dos elementos presentes nas páginas que acessa ao longo do registro do *click*. Isso inclui elementos como o "favicon.ico", ícone pedido pelo navegador em páginas web, e a imagem do *ad* em si, que deveria ser carregada logo depois do pedido do *Javascript* que carrega o *ad* no site publicador.

Vários desses módulos foram implementados com conhecimento que um atacante normalmente não teria, ou que não estariam implementados em um robô genérico de *clicks*,

como saber que seus *clicks* estão sendo bloqueados por estarem sendo enviados com intervalos de tempo constantes.

Capítulo 5

Testes e Resultados

Para testar o sistema, simplesmente utilizamos o *bot* desenvolvido contra o sistema de defesa, com combinações diferentes de módulos habilitados, e por fim com todos ativos ao mesmo tempo. O sistema conseguiu identificar a fraude na maioria dos os casos, com a exceção dos ataques em baixa frequência. Esse tipo de ataque, como já dito, requer uma análise mais detalhada de padrões de acesso, e é difícil de categorizar de forma automática. A atividade de fraude do *bot* com os outros módulos ativos, porém, foi detectada pelo sistema com sucesso.

Os testes foram realizados no sistema operacional Windows 10. Os servidores que representam os 3 principais agentes da publicidade online e o bot todos foram rodados em rede local, usando diferentes portas para simular os diferentes sites envolvidos no estudo. Os servidores dos 3 agentes se comportam de maneira normal, podendo servir a qualquer *request* do tipo *GET* recebido, enquanto o *bot* é chamado 5 vezes, cada vez com uma configuração diferente, definida por arquivos simples que apenas dizem se certos módulos estão ativados ou desativados. As diferentes configurações tem o intúito de representar o nível de complexidade do robô. A tabela 5.1 define as diferentes combinações que foram usadas, e nela '1' significa que o módulo está ativo, e '0' é seu desativamento. Os números romanos indicam as diferentes configurações. Para as configurações do robô, também estamos utilizando 3 *threads* em paralelo para cada tentativa de click feito. Devido à maneira como o *click* é registrado, cada acesso em paralelo desses é contado como um novo *click*, e portanto pode ser benificial para o atacante.

Já na tabela 5.2, temos os valores dos pesos utilizados para o teste do sistema. Precisamos relembrar que esses valores não são finais nem obrigatórios para o funcionamento do sistema, e apenas seguem o que acreditamos serem boas razões relativas entre si, ou seja, regras que identificam um usuário malicioso com mais certeza devem ter peso maior proporcionalmente. Lembramos também que regras com pesos negativos só podem contar positivamente para o placar final.

Módulo	**I**	**II**	**III**	**IV**	**V**	**VI**
NormalAccessModule	1	1	1	1	1	1
HumanTimerModule	1	0	1	1	1	1
HeadersModule	0	1	1	1	1	1
DNTModule	0	1	1	1	1	1
WideAccessModule	0	0	1	1	1	1
SelectiveAccessModule	0	0	0	1	1	1
CookieModule	0	0	0	0	1	1
CompleteAccessModule	0	0	0	0	0	1
RandomTimeModule	0	0	0	0	0	1

Tabela 5.1: Configurações de teste para o *bot*.

Regra	Peso
JavascriptEnabledRule	2.0
UserAgentRule	2.0
RedirectTimeRule	3.0
DNTRule	-1.0
BehaviorRule	3.0
TimePeriodRule	2.0

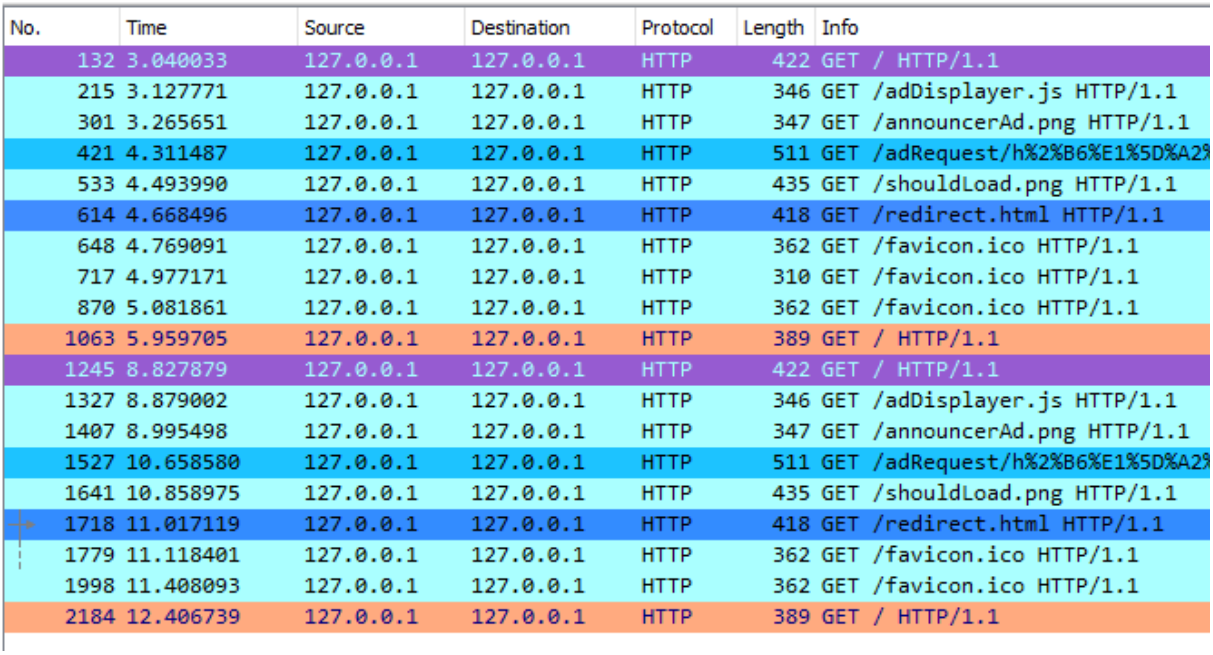
Tabela 5.2: Pesos escolhidos para as regras indicativas.

Antes de testar a defesa do sistema contra os vários níveis de ataque, devemos avaliar se o sistema opera como previsto se visitado por um usuário normal. Para essa parte do teste, os sites foram todos percorridos duas vezes com o navegador *Firefox* 59.0.1 (64-bit). Na imagem 5.1 temos a captura de tráfego local que mostra os pacotes enviados ao longo da conversa. Nas imagems de tráfego, temos o seguinte esquema de cores:

- A cor roxa representa o site do publicador;
- Os azuis mostram a rede de anúncios;
- Os tons mais escuros de azul mostram as requisições que representam *clicks*, e;
- O anunciante é mostrado com a cor laranja.

Na tabela 5.3 temos os detalhes de alguns dos campos enviados para o servidor em cada troca *HTTP*. O relatório de segurança gerado pode ser visto em 5.4, e mostra que o sistema não acusou o usuário legítimo de fraude, ou seja, o teste foi um sucesso.

No primeiro ataque, vemos que os *bots* são identificados como fraudes logo no começo do processo do sistema. Já que nessa configuração os *headers* não são alterados para valores normais, como visto no campo *user_agent* da tabela 5.5, então é possível ver que não são usuários comums logo no começo. A imagem 5.2 mostra o tráfego desse ataque, e a tabela 5.6 reporta os resultados de cada *bot* no sistema. Note que o muitas regras não



No.	Time	Source	Destination	Protocol	Length	Info
132	3.040033	127.0.0.1	127.0.0.1	HTTP	422	GET / HTTP/1.1
215	3.127771	127.0.0.1	127.0.0.1	HTTP	346	GET /adDisplayer.js HTTP/1.1
301	3.265651	127.0.0.1	127.0.0.1	HTTP	347	GET /announcerAd.png HTTP/1.1
421	4.311487	127.0.0.1	127.0.0.1	HTTP	511	GET /adRequest/h%2%B6%E1%5D%A2%
533	4.493990	127.0.0.1	127.0.0.1	HTTP	435	GET /shouldLoad.png HTTP/1.1
614	4.668496	127.0.0.1	127.0.0.1	HTTP	418	GET /redirect.html HTTP/1.1
648	4.769091	127.0.0.1	127.0.0.1	HTTP	362	GET /favicon.ico HTTP/1.1
717	4.977171	127.0.0.1	127.0.0.1	HTTP	310	GET /favicon.ico HTTP/1.1
870	5.081861	127.0.0.1	127.0.0.1	HTTP	362	GET /favicon.ico HTTP/1.1
1063	5.959705	127.0.0.1	127.0.0.1	HTTP	389	GET / HTTP/1.1
1245	8.827879	127.0.0.1	127.0.0.1	HTTP	422	GET / HTTP/1.1
1327	8.879002	127.0.0.1	127.0.0.1	HTTP	346	GET /adDisplayer.js HTTP/1.1
1407	8.995498	127.0.0.1	127.0.0.1	HTTP	347	GET /announcerAd.png HTTP/1.1
1527	10.658580	127.0.0.1	127.0.0.1	HTTP	511	GET /adRequest/h%2%B6%E1%5D%A2%
1641	10.858975	127.0.0.1	127.0.0.1	HTTP	435	GET /shouldLoad.png HTTP/1.1
1718	11.017119	127.0.0.1	127.0.0.1	HTTP	418	GET /redirect.html HTTP/1.1
1779	11.118401	127.0.0.1	127.0.0.1	HTTP	362	GET /favicon.ico HTTP/1.1
1998	11.408093	127.0.0.1	127.0.0.1	HTTP	362	GET /favicon.ico HTTP/1.1
2184	12.406739	127.0.0.1	127.0.0.1	HTTP	389	GET / HTTP/1.1

Figura 5.1: Tráfego HTTP de 2 acessos ao *ad* via *browser*.

id	*time*	*path*	*user_agent*	*cookies*	*status*	*report_id*
9	10:48:22.067621	/adDisplayer.js	Mozilla/5.0(...)		valid	
10	10:48:22.183617	/announcerAd.png	Mozilla/5.0(...)		valid	
12	10:48:23.8467	/adRequest.html	Mozilla/5.0(...)		valid	2
11	10:48:24.047093	/shouldLoad.png	Mozilla/5.0(...)		valid	
14	10:48:24.205252	/redirect.html	Mozilla/5.0(...)	JSEnabled=true	valid	2
13	10:48:24.306518	/favicon.ico	Mozilla/5.0(...)		valid	
15	10:48:24.598219	/favicon.ico	Mozilla/5.0(...)		valid	

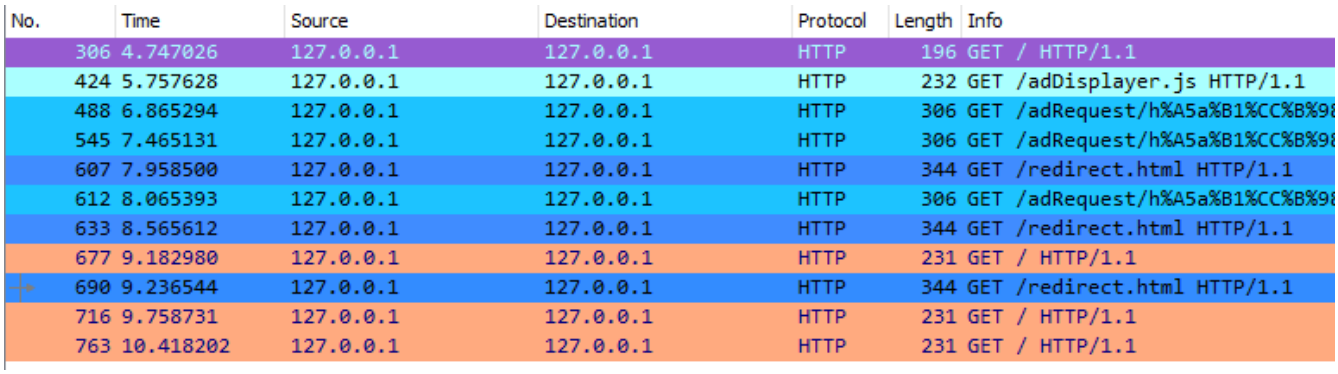
Tabela 5.3: *Requests* para o navegador.

id	*dnt*	*java*	*human*	*list*	*agent*	*redir*	*lang*	*behav*	*pages*	*tperiod*	*score*
2	t	t	t	t	t	t	t				1.17

Tabela 5.4: Resultado das regras para o navegador.

foram acionadas, já que cada usuário em paralelo foi determinado como malicioso logo no primeiro acesso.

Com a segunda configuração do robô, vemos novamente que a maioria das tentativas de ataque foram descobertas na primeira rodada, como visto na tabela 5.8. O primeiro *bot* conseguiu passar da primeira página sem ser identificado como fraude, mas falhou nas regras **JavascripRule** e **RedirectTime** logo depois. Os outros dois *bots* foram identificados na primeira rodada, já que a falta do módulo **HumanTimerModule** acionou a regra **HumanTimerRule**. O valor de tempo entre os pedidos foi de aprocimadamente 0.2 segundos, como visto na tabela 5.7. O primeiro *bot* passou por essa regra devido a



No.	Time	Source	Destination	Protocol	Length	Info
306	4.747026	127.0.0.1	127.0.0.1	HTTP	196	GET / HTTP/1.1
424	5.757628	127.0.0.1	127.0.0.1	HTTP	232	GET /adDisplayer.js HTTP/1.1
488	6.865294	127.0.0.1	127.0.0.1	HTTP	306	GET /adRequest/h%A5a%B1%CC%B%9
545	7.465131	127.0.0.1	127.0.0.1	HTTP	306	GET /adRequest/h%A5a%B1%CC%B%9
607	7.958500	127.0.0.1	127.0.0.1	HTTP	344	GET /redirect.html HTTP/1.1
612	8.065393	127.0.0.1	127.0.0.1	HTTP	306	GET /adRequest/h%A5a%B1%CC%B%9
633	8.565612	127.0.0.1	127.0.0.1	HTTP	344	GET /redirect.html HTTP/1.1
677	9.182980	127.0.0.1	127.0.0.1	HTTP	231	GET / HTTP/1.1
690	9.236544	127.0.0.1	127.0.0.1	HTTP	344	GET /redirect.html HTTP/1.1
716	9.758731	127.0.0.1	127.0.0.1	HTTP	231	GET / HTTP/1.1
763	10.418202	127.0.0.1	127.0.0.1	HTTP	231	GET / HTTP/1.1

Figura 5.2: Tráfego HTTP do *bot* com configuração 1.

id	*time*	*path*	*user_agent*	*cookies*	*status*	*report_id*
16	11:40:01.66616	/adDisplayer.js	python-requ(...)		valid	
17	11:40:02.773198	/adRequest.html	python-requ(...)		fraud	3
19	11:40:03.372003	/adRequest.html	python-requ(...)		fraud	4
18	11:40:03.865374	/redirect.html	python-requ(...)	JSEnabled=true;	fraud	3
21	11:40:03.973241	/adRequest.html	python-requ(...)		fraud	5
20	11:40:04.472487	/redirect.html	python-requ(...)	JSEnabled=true;	fraud	4
22	11:40:05.142415	/redirect.html	python-requ(...)	JSEnabled=true;	fraud	5

Tabela 5.5: *Requests* da configuração 1.

id	*dnt*	*java*	*human*	*list*	*agent*	*redir*	*lang*	*behav*	*pages*	*tperiod*	*score*
3	f		t	t			f				0.00
4	f		t	t			f				0.00
5	f		t	t			f				0.00

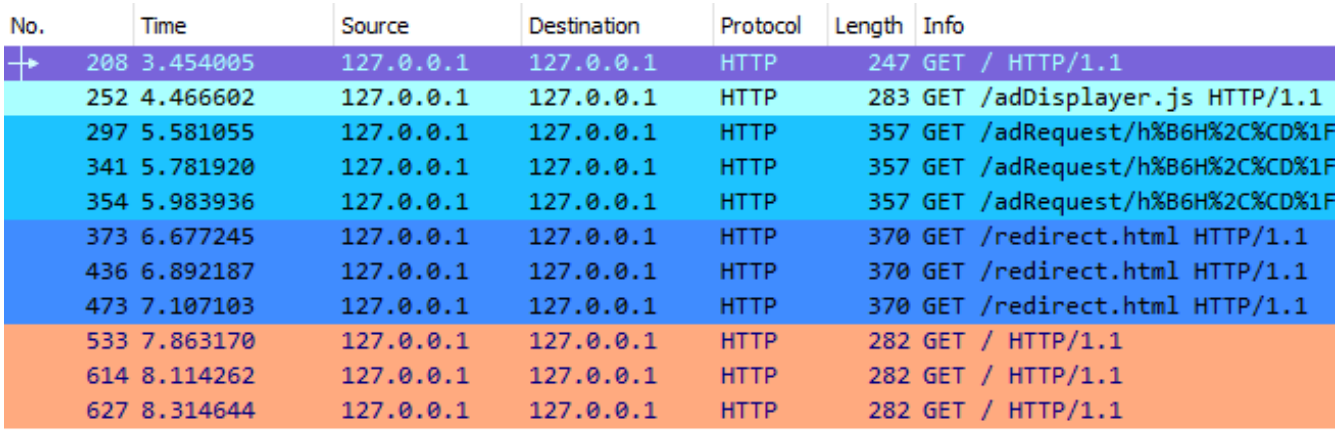
Tabela 5.6: Resultado das regras para a configuração 1.

não haverem requisições recentes do mesmo IP antes dele. O tráfego pode ser visualizado na imagem 5.3

id	*time*	*path*	*user_agent*	*cookies*	*status*	*report_id*
23	13:04:31.049619	/adDisplayer.js	Chrome/41.0(...)		valid	
24	13:04:32.161772	/adRequest.html	Chrome/41.0(...)		fraud	6
26	13:04:32.363818	/adRequest.html	Chrome/41.0(...)		fraud	7
28	13:04:32.564822	/adRequest.html	Chrome/41.0(...)		fraud	8
25	13:04:33.258126	/redirect.html	Chrome/41.0(...)		fraud	6
27	13:04:33.473077	/redirect.html	Chrome/41.0(...)		fraud	7
29	13:04:33.687692	/redirect.html	Chrome/41.0(...)		fraud	8

Tabela 5.7: *Requests* da configuração 2.

Para a terceira versão do ataque, o robô utiliza os módulos das duas últimas versões e adiciona o módulo **WideAccessModule**. O efeito dessa mudança é visto na imagem 5.4 e na tabela 5.9: o *bot* agora acessa as imagens contidas dentro das páginas transi-



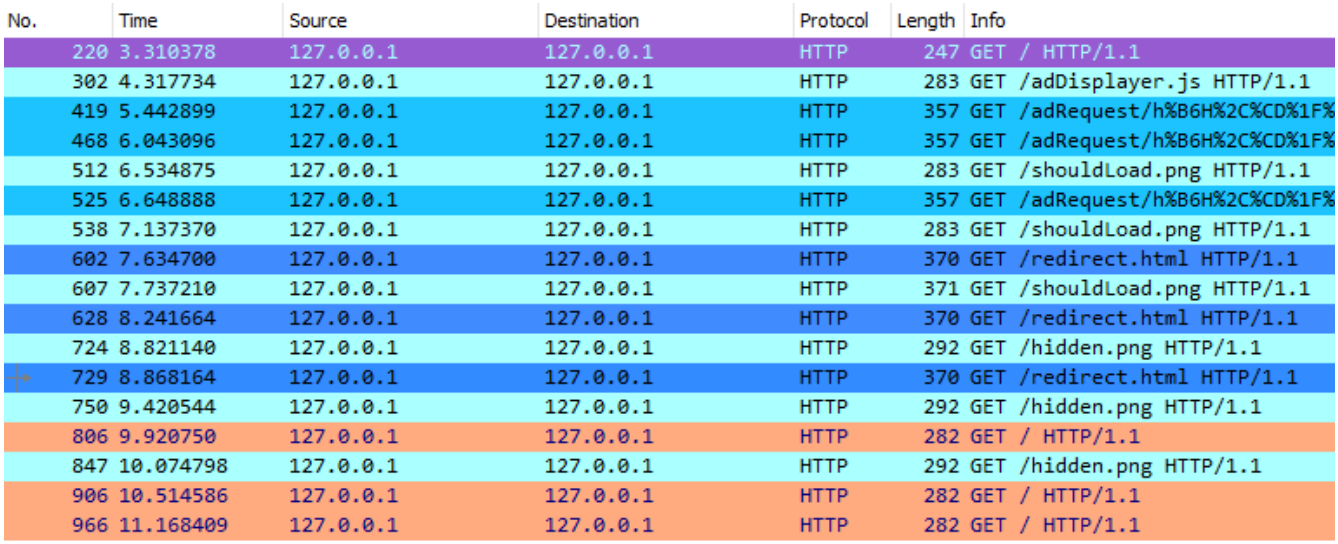
No.	Time	Source	Destination	Protocol	Length	Info
208	3.454005	127.0.0.1	127.0.0.1	HTTP	247	GET / HTTP/1.1
252	4.466602	127.0.0.1	127.0.0.1	HTTP	283	GET /adDisplayer.js HTTP/1.1
297	5.581055	127.0.0.1	127.0.0.1	HTTP	357	GET /adRequest/h%B6H%2C%CD%1F
341	5.781920	127.0.0.1	127.0.0.1	HTTP	357	GET /adRequest/h%B6H%2C%CD%1F
354	5.983936	127.0.0.1	127.0.0.1	HTTP	357	GET /adRequest/h%B6H%2C%CD%1F
373	6.677245	127.0.0.1	127.0.0.1	HTTP	370	GET /redirect.html HTTP/1.1
436	6.892187	127.0.0.1	127.0.0.1	HTTP	370	GET /redirect.html HTTP/1.1
473	7.107103	127.0.0.1	127.0.0.1	HTTP	370	GET /redirect.html HTTP/1.1
533	7.863170	127.0.0.1	127.0.0.1	HTTP	282	GET / HTTP/1.1
614	8.114262	127.0.0.1	127.0.0.1	HTTP	282	GET / HTTP/1.1
627	8.314644	127.0.0.1	127.0.0.1	HTTP	282	GET / HTTP/1.1

Figura 5.3: Tráfego HTTP do *bot* com configuração 2.

id	*dnt*	*java*	*human*	*list*	*agent*	*redir*	*lang*	*behav*	*pages*	*tperiod*	*score*
6	t	f	t	t	t	f	t				0.43
7	t		f	t			t				0.00
8	t		f	t			t				0.00

Tabela 5.8: Resultado das regras para a configuração 2.

tórias de redirecionamento, incluindo a imagem escondida no código de "redirect.html", "hidden.png". Apesar das mudanças, os *bots* ainda falham nos testes *online* 5.10.



No.	Time	Source	Destination	Protocol	Length	Info
220	3.310378	127.0.0.1	127.0.0.1	HTTP	247	GET / HTTP/1.1
302	4.317734	127.0.0.1	127.0.0.1	HTTP	283	GET /adDisplayer.js HTTP/1.1
419	5.442899	127.0.0.1	127.0.0.1	HTTP	357	GET /adRequest/h%B6H%2C%CD%1F%
468	6.043096	127.0.0.1	127.0.0.1	HTTP	357	GET /adRequest/h%B6H%2C%CD%1F%
512	6.534875	127.0.0.1	127.0.0.1	HTTP	283	GET /shouldLoad.png HTTP/1.1
525	6.648888	127.0.0.1	127.0.0.1	HTTP	357	GET /adRequest/h%B6H%2C%CD%1F%
538	7.137370	127.0.0.1	127.0.0.1	HTTP	283	GET /shouldLoad.png HTTP/1.1
602	7.634700	127.0.0.1	127.0.0.1	HTTP	370	GET /redirect.html HTTP/1.1
607	7.737210	127.0.0.1	127.0.0.1	HTTP	371	GET /shouldLoad.png HTTP/1.1
628	8.241664	127.0.0.1	127.0.0.1	HTTP	370	GET /redirect.html HTTP/1.1
724	8.821140	127.0.0.1	127.0.0.1	HTTP	292	GET /hidden.png HTTP/1.1
729	8.868164	127.0.0.1	127.0.0.1	HTTP	370	GET /redirect.html HTTP/1.1
750	9.420544	127.0.0.1	127.0.0.1	HTTP	292	GET /hidden.png HTTP/1.1
806	9.920750	127.0.0.1	127.0.0.1	HTTP	282	GET / HTTP/1.1
847	10.074798	127.0.0.1	127.0.0.1	HTTP	292	GET /hidden.png HTTP/1.1
906	10.514586	127.0.0.1	127.0.0.1	HTTP	282	GET / HTTP/1.1
966	11.168409	127.0.0.1	127.0.0.1	HTTP	282	GET / HTTP/1.1

Figura 5.4: Tráfego HTTP do *bot* com configuração 3.

Ao verificar a tabela de resultados 5.12, notamos que os placares são similares ao da configuração anterior. Apesar da mudança visível na captura de tráfego 5.5 e na tabela 5.11, nas quais o robô não acessa mais a imagem escondida dentro da página da rede de

id	*time*	*path*	*user_agent*	*cookies*	*status*	*report_id*
30	13:21:55.777138	/adDisplayer.js	Chrome/41.0(...)		valid	
33	13:21:56.90226	/adRequest.html	Chrome/41.0(...)		fraud	9
36	13:21:57.501496	/adRequest.html	Chrome/41.0(...)		fraud	10
31	13:21:57.993412	/shouldLoad.png	Chrome/41.0(...)		valid	
39	13:21:58.10729	/adRequest.html	Chrome/41.0(...)		fraud	11
32	13:21:58.595796	/shouldLoad.png	Chrome/41.0(...)		valid	
34	13:21:59.093088	/redirect.html	Chrome/41.0(...)		fraud	9
35	13:21:59.196603	/shouldLoad.png	Chrome/41.0(...)		valid	
37	13:21:59.700085	/redirect.html	Chrome/41.0(...)		fraud	10
38	13:22:00.28054	/hidden.png	Chrome/41.0(...)		valid	
40	13:22:00.326873	/redirect.html	Chrome/41.0(...)		fraud	11
41	13:22:00.878942	/hidden.png	Chrome/41.0(...)		valid	
42	13:22:01.533731	/hidden.png	Chrome/41.0(...)		valid	

Tabela 5.9: *Requests* da configuração 3.

id	*dnt*	*java*	*human*	*list*	*agent*	*redir*	*lang*	*behav*	*pages*	*tperiod*	*score*
9	t	f	t	t	t	f	t				0.43
10	t	f	t	t	t	f	t				0.43
11	t	f	t	t	t	f	t				0.43

Tabela 5.10: Resultado das regras para a configuração 3.

anúncios, ele ainda é visto como sendo malicioso ao falhar nas regras **JavascriptEnabledRule** e **RedirectTimeRule**.

No.	Time	Source	Destination	Protocol	Length	Info
147	3.401253	127.0.0.1	127.0.0.1	HTTP	247	GET / HTTP/1.1
247	4.413518	127.0.0.1	127.0.0.1	HTTP	283	GET /adDisplayer.js HTTP/1.1
447	5.521711	127.0.0.1	127.0.0.1	HTTP	357	GET /adRequest/h%B6H%2C%CD%1F%
550	6.121653	127.0.0.1	127.0.0.1	HTTP	357	GET /adRequest/h%B6H%2C%CD%1F%
605	6.617088	127.0.0.1	127.0.0.1	HTTP	283	GET /shouldLoad.png HTTP/1.1
618	6.722160	127.0.0.1	127.0.0.1	HTTP	357	GET /adRequest/h%B6H%2C%CD%1F%
651	7.218204	127.0.0.1	127.0.0.1	HTTP	283	GET /shouldLoad.png HTTP/1.1
724	7.709472	127.0.0.1	127.0.0.1	HTTP	370	GET /redirect.html HTTP/1.1
729	7.826211	127.0.0.1	127.0.0.1	HTTP	371	GET /shouldLoad.png HTTP/1.1
790	8.309394	127.0.0.1	127.0.0.1	HTTP	370	GET /redirect.html HTTP/1.1
850	8.909023	127.0.0.1	127.0.0.1	HTTP	282	GET / HTTP/1.1
879	8.955523	127.0.0.1	127.0.0.1	HTTP	370	GET /redirect.html HTTP/1.1
910	9.488783	127.0.0.1	127.0.0.1	HTTP	282	GET / HTTP/1.1
1034	10.137859	127.0.0.1	127.0.0.1	HTTP	282	GET / HTTP/1.1

Figura 5.5: Tráfego HTTP do *bot* com configuração 4.

A partir da configuração 5 do *bot*, vemos que a análise online sozinha não é capaz de identificar a fraude, como mostrado pelo relatório 5.14. Nessa versão o bot já utiliza uma análise específica do *software* envolvido na rede de anúncios, e contorna essas técnicas de defesa, com exceção da regra **RedirectTimeRule**. Dessa forma, precisamos chamar as

id	*time*	*path*	*user_agent*	*cookies*	*status*	*report_id*
43	16:55:27.597077	/adDisplayer.js	Chrome/41.0(...)		valid	
46	16:55:28.703283	/adRequest.html	Chrome/41.0(...)		fraud	12
49	16:55:29.303221	/adRequest.html	Chrome/41.0(...)		fraud	13
44	16:55:29.798657	/shouldLoad.png	Chrome/41.0(...)		valid	
51	16:55:29.904731	/adRequest.html	Chrome/41.0(...)		fraud	14
45	16:55:30.399783	/shouldLoad.png	Chrome/41.0(...)		valid	
47	16:55:30.891066	/redirect.html	Chrome/41.0(...)		fraud	12
48	16:55:31.008756	/shouldLoad.png	Chrome/41.0(...)		valid	
50	16:55:31.490979	/redirect.html	Chrome/41.0(...)		fraud	13
52	16:55:32.137027	/redirect.html	Chrome/41.0(...)		fraud	14

Tabela 5.11: *Requests* da configuração 4.

id	*dnt*	*java*	*human*	*list*	*agent*	*redir*	*lang*	*behav*	*pages*	*tperiod*	*score*
12	t	f	t	t	t	f	t				0.43
13	t	f	t	t	t	f	t				0.43
14	t	f	t	t	t	f	t				0.43

Tabela 5.12: Resultado das regras para a configuração 4.

regras de análise offline para chegar ao resultado correto, e isso é verificado no relatório 5.15. Notamos que os *clicks* são identificados como fraudes tanto pela regra **PagesLoadedRule** quanto pela regra **TimePeriodRule**. O resultado final no banco de dados pode ser visto na tabela 5.13

No.	Time	Source	Destination	Protocol	Length	Info
197	2.886838	127.0.0.1	127.0.0.1	HTTP	247	GET / HTTP/1.1
283	3.893007	127.0.0.1	127.0.0.1	HTTP	283	GET /adDisplayer.js HTTP/1.1
351	5.020273	127.0.0.1	127.0.0.1	HTTP	357	GET /adRequest/h%B6H%2C%CD%1F
388	5.620293	127.0.0.1	127.0.0.1	HTTP	357	GET /adRequest/h%B6H%2C%CD%1F
433	6.113186	127.0.0.1	127.0.0.1	HTTP	283	GET /shouldLoad.png HTTP/1.1
446	6.221808	127.0.0.1	127.0.0.1	HTTP	357	GET /adRequest/h%B6H%2C%CD%1F
459	6.723039	127.0.0.1	127.0.0.1	HTTP	283	GET /shouldLoad.png HTTP/1.1
539	7.217641	127.0.0.1	127.0.0.1	HTTP	395	GET /redirect.html HTTP/1.1
544	7.315458	127.0.0.1	127.0.0.1	HTTP	371	GET /shouldLoad.png HTTP/1.1
565	7.817517	127.0.0.1	127.0.0.1	HTTP	395	GET /redirect.html HTTP/1.1
609	8.407001	127.0.0.1	127.0.0.1	HTTP	282	GET / HTTP/1.1
622	8.458127	127.0.0.1	127.0.0.1	HTTP	395	GET /redirect.html HTTP/1.1
655	9.001575	127.0.0.1	127.0.0.1	HTTP	282	GET / HTTP/1.1
699	9.638614	127.0.0.1	127.0.0.1	HTTP	282	GET / HTTP/1.1

Figura 5.6: Tráfego HTTP do *bot* com configuração 5.

Para finalizar o teste, analisamos como o sistema se saiu contra a configuração 6. Já sabemos a partir do últimos teste que apenas a análise online não foi o bastante para encontrar as fraudes. No resultado das regras offline e online em 5.17, vemos que dessa vez o *bot* consegue passar pelo teste **PagesLoadedRule** graças ao módulo **CompleteAccessModule**, mas ainda falha contra os testes **TimePeriodRule** e **RedirectTimeRule**.

id	*time*	*path*	*user_agent*	*cookies*	*status*	*report_id*
53	17:06:56.681019	/adDisplayer.js	Chrome/41.0(...)		valid	
56	17:06:57.807404	/adRequest.html	Chrome/41.0(...)		fraud	15
59	17:06:58.407286	/adRequest.html	Chrome/41.0(...)		fraud	16
54	17:06:58.900208	/shouldLoad.png	Chrome/41.0(...)		valid	
61	17:06:59.00882	/adRequest.html	Chrome/41.0(...)		fraud	17
55	17:06:59.511064	/shouldLoad.png	Chrome/41.0(...)		valid	
57	17:07:00.004654	/redirect.html	Chrome/41.0(...)	JSEnabled=true;	fraud	15
58	17:07:00.103444	/shouldLoad.png	Chrome/41.0(...)		valid	
60	17:07:00.604711	/redirect.html	Chrome/41.0(...)	JSEnabled=true;	fraud	16
62	17:07:01.246168	/redirect.html	Chrome/41.0(...)	JSEnabled=true;	fraud	17

Tabela 5.13: *Requests* da configuração 5.

id	*dnt*	*java*	*human*	*list*	*agent*	*redir*	*lang*	*behav*	*pages*	*tperiod*	*score*
15	t	t	t	t	t	f	t				0.71
16	t	t	t	t	t	f	t				0.71
17	t	t	t	t	t	f	t				0.71

Tabela 5.14: Resultado das regras para a configuração 5.

id	*dnt*	*java*	*human*	*list*	*agent*	*redir*	*lang*	*behav*	*pages*	*tperiod*	*score*
15	t	t	t	t	t	f	t		f	f	0.42
16	t	t	t	t	t	f	t		f	f	0.42
17	t	t	t	t	t	f	t		f	f	0.42

Tabela 5.15: Resultado das regras *online* e regras *offline* para a configuração 5.

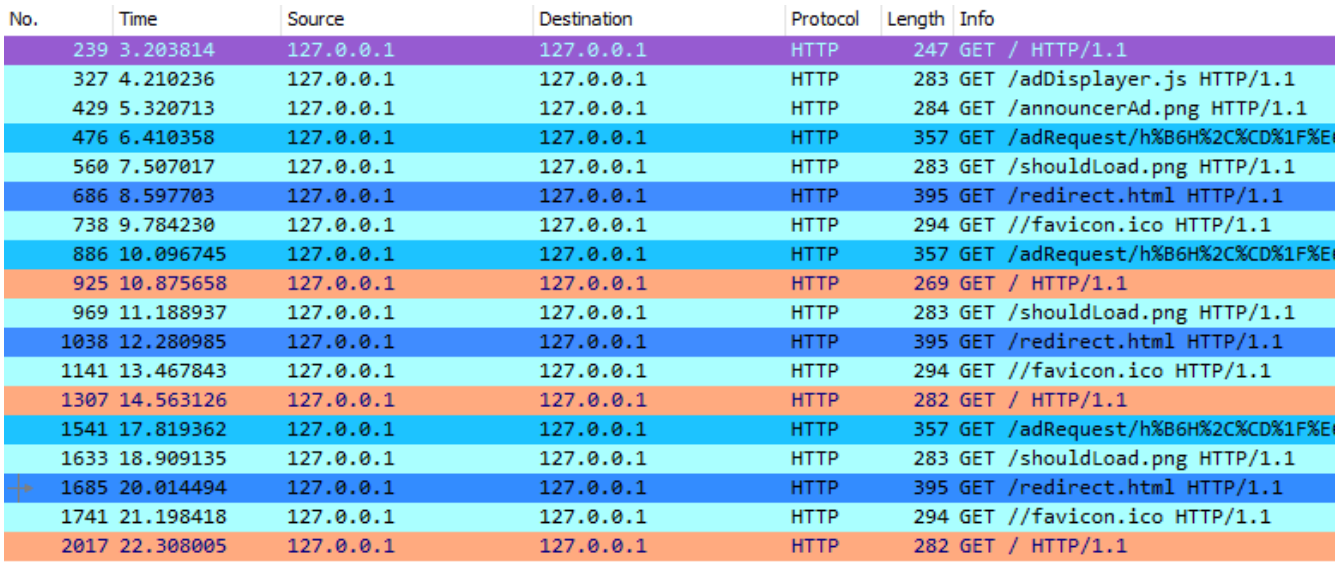


Figura 5.7: Tráfego HTTP do *bot* com configuração 6.

Vemos por fim que o sistema conseguiu detectar todos os tipos de fraude apresentados. Apesar de algumas configurações, principalmente a 6, conseguirem passar na maioria

id	*time*	*path*	*user_agent*	*cookies*	*status*	*report_id*
63	17:12:36.13271	/adDisplayer.js	Chrome/41.0(...)		valid	
64	17:12:37.242392	/announcerAd.png	Chrome/41.0(...)		valid	
66	17:12:38.332039	/adRequest.html	Chrome/41.0(...)		fraud	18
65	17:12:39.428698	/shouldLoad.png	Chrome/41.0(...)		valid	
67	17:12:40.520384	/redirect.html	Chrome/41.0(...)	JSEnabled=true;	fraud	18
68	17:12:41.70599	//favicon.ico	Chrome/41.0(...)		valid	
70	17:12:42.018585	/adRequest.html	Chrome/41.0(...)		fraud	19
69	17:12:43.110647	/shouldLoad.png	Chrome/41.0(...)		valid	
71	17:12:44.202571	/redirect.html	Chrome/41.0(...)	JSEnabled=true;	fraud	19
72	17:12:45.38956	//favicon.ico	Chrome/41.0(...)		valid	
74	17:12:49.741072	/adRequest.html	Chrome/41.0(...)		fraud	20
73	17:12:50.830796	/shouldLoad.png	Chrome/41.0(...)		valid	
75	17:12:51.936176	/redirect.html	Chrome/41.0(...)	JSEnabled=true;	fraud	20
76	17:12:53.120084	//favicon.ico	Chrome/41.0(...)		valid	

Tabela 5.16: *Requests* da configuração 6.

id	*dnt*	*java*	*human*	*list*	*agent*	*redir*	*lang*	*behav*	*pages*	*tperiod*	*score*
18	t	t	t	t	t	f	t		t	f	0.42
19	t	t	t	t	t	f	t		t	f	0.42
20	t	t	t	t	t	f	t		t	f	0.42

Tabela 5.17: Resultado das regras *online* e regras *offline* para a configuração 5.

dos testes, os pesos altos das regras **RedirectTimeRule** e **TimePeriodRule** foram o sufiente para impedir a fraude prolongada contra o sistema. É importante notar então que os pesos diferentes alteram como o sistema responderia a diferentes tipos de fraud: se **RedirectTimeRule** tivesse peso 2 em vez de 3 e **TimePeriodRule** um peso de 1 ao em vez de 2, a configuração 6 teria passado pelo sistema sem ser percebida.

Capítulo 6

Conclusões

Os resultados obtidos nos testes bateram com o desempenho esperado do sistema. Todas as *click frauds* foram identificadas, porém uma configuração diferente de pesos para as regras poderia levar a resultados drasticamente diferentes, então resaltamos que é necessário escolher cuidadosamente esses valores. Os pesos apresentados aqui não são os únicos que funcionariam, tão pouco são otimizados para a melhor detecção possível. O sistema, apesar de não ter sido ainda testado contra um ambiente real, como os resultados de Xu et. al [1] ou de Kitts et. al [6], apresenta bom desempenho contra as técnicas de fraude aqui vistas e simuladas, sempre disparando no mínimo duas regras para cada *click* malicioso utilizado.

O sistema ainda assim é vulnerável a ataques a longo prazo, ou ataques de baixa frequência. Cliques maliciosos vindos em intervalos grandes de tempo provavelmente não serão detectados automaticamente pelo sistema, e se cometidos em larga escala, com endereços IP diversos agindo sempre em tempos diversos, podem causar problemas aos agentes envolvidos, mas existe uma barreira técnica considerável para obter uma massa considerável de IPs para cometer esse tipo de ataque e receber um retorno considerável. Versões mais simples ou menos agressivas desse mesmo ataque podem ser economicamente inviáveis, devido ao baixo retorno por um único clique.

É importante, apesar do funcionamento do sistema elaborado ao longo do trabalho, lembrar que existem desafios para o estudo e desenvolvimento de sistemas de defesa contra *click fraud* em geral, um dos maiores sendo a falta de massas de dado utilizáveis e acessíveis publicamente, que fere a possibilidade de implementar sistemas com o uso de *machine learning* já treinados contra vários tipos de fraude previamente conhecidos. Outro problema, esse exclusivo para a análise de *click frauds* baseada apenas na interação da *ad network* no processo de anúncios virtuais, é a falta de acesso ao comportamento do usuário nos sites que são de fato acessados por ele, e nos quais se espera ver um comportamento humano. Em muitos casos, capturar dados como tempo de acesso em

cada página do domínio e movimento do mouse é um elemento essencial para método de detecção mais avançados, já que demonstram padrões inerentemente humanos e de certa forma aleatórios, que dificilmente serão reproduzidos por acessos artificiais aos mesmos *sites*.

6.1 Trabalhos Futuros

6.1.1 Regra *LoadingBehaviorRule*

Essa regra utilizaria os dados de comportamento do usuário coletados durante o tempo em que seu navegador passa pelas páginas hospedadas pela rede de anúncios, e decide então se esse comportamento parece ser natural ou de um robô. Esse teste, assim como o *DoNotTrackRule*, deveria um peso negativo, ou seja, passar nele ajuda o usuário a ser certificado como legítimo, mas falhar nesse teste não é prejudicial.

Para armazenar as informações relevantes sobre o acesso do usuário, propomos um relatório mais simples que o feito para a regra *ExternalBehaviorRule*, como visto na tabela 6.1.

Categoria	**Campos**
Mouse Moves	Na página inicial
	Na segunda página
Mouse Events	Na página inicial
	Na segunda página

Tabela 6.1: Relatório de comportamento do usuário no carregamento das páginas

6.1.2 Detecção de Funcionalidade de *Browser*

Um possível recurso a ser considerado e implementado no sistema proposto é a detecção de funcionalidades de *browser*. Lembramos que, como já mencionado antes, *bots* em geral não são implementados como navegadores. Uma das formas mais usadas de *click fraud* involve a infiltração no computador pessoal de usuários comuns. Depois de infectado, o computador poderá ter algum programa malicioso rodando em *background*, que fica à espera de um servidor principal, e quando recebe a instrução, usa o computador do usuário inocente para praticar a fraude. Eles conseguem fazer isso por serem códigos leves e rápidos, podendo ser executados sem alterações perceptiveis ao usuário.

Um teste de detecção de funcionalidade de browser tiraria proveito desse design de *bot*, assim como outros já no sistema, e faria uso de uma lista de funcionalidades consideradas universais em navegadores recentes, como a feita em 2014 por H. Xu et al. [1], em que

são apresentadas as funcionalidades em comum entre os 5 navegadores mais utilizados (nominalmente, *Chrome*, *Firefox*, *Internet Explorer*, *Safari* e *Opera*).

6.1.3 *Browser Fingerprinting*

Outro campo moderno da computação que pode ser aplicado futuramente ao sistema é o de *Browser Fingerprinting*, em que são feitos alguns pedidos do usuário, mais especificamente para seu *browser*, como pedir para que ele renderize um objeto em três-dimensões. É possível obter algumas características únicas que diferem entre *browsers* e usuário diferentes. Isso dá ao sistema uma maneira adicional para identificar cada usuário único, e pode ajudar a identififcar *bots* que usam *browsers* diretamente para seus ataques.

Relacionado com *Browser Fingerprinting*, existe também o campo de *System Fingerprinting*, que tem um foco similar, porém para sistemas operacionais em vez de navegadores. Esses métodos podem identificar um sistema de forma única, podendo então ser usado em conjunto com as técnicas já apresentadas até agora para ajudar a identificar até fraudes complexas que vêm de um computador que executa todo seu esquema em um único sistema operacional, por exemplo.

6.1.4 *Machine Learning*

Técnicas de *machine learning* podem ser usadas para otimizar e automatizar certas partes do sistema proposto, como o peso das regras utilizadas para julgar os *clicks* de um usuário. O aprendizado de máquina poderia usar *clicks* já capturados e analisados pela equipe de segurança responsável como dados de treinamento, e depois de treinado o sistema conseguiria gerênciar os pesos e combinações de regras que melhor identificam usuário maliciosos.

6.1.5 Aplicação no Mundo Real

Apesar dos cuidados tomados na elaboração desse estudo, sempre existem detalhes inesperados ou não considerados quando uma aplicação é colocada para funcionar em um ambiente real. Questões como escalabilidade e manutenabilidade passam a ser essenciais para o bom funcionamento do sistema. Além disso, como estamos tratando de um sistema de segurança, é necessário considerar futuros ataques e futuras defesas, que podem vir a serem postos contra as técnicas já propostas.

Referências

[1] Xu, Haitao, Daiping Liu, Aaron Koehl, Haining Wang e Angelos Stavrou: *Click fraud detection on the advertiser side*. Em *European Symposium on Research in Computer Security*, páginas 419–438. Springer, 2014. ix, 8, 9, 11, 16, 18, 35, 36

[2] *Reaction time statistics*. `https://www.humanbenchmark.com/tests/reactiontime/statistics`. ix, 15

[3] McNair, Corey: *Us ad spending: emarketer's updated estimates and forecast for 2017*, setembro 2017. 1

[4] Fielding, R. e J. Reschke: *Hypertext transfer protocol (http/1.1): Semantics and content*. Rfc 7231, RFC Editor, June 2014. `http://www.rfc-editor.org/rfc/rfc7231.txt`, `http://www.rfc-editor.org/rfc/rfc7231.txt`. 7

[5] Fielding, R. e J. Reschke: *Hypertext transfer protocol (http/1.1): Authentication*. Rfc 7235, RFC Editor, June 2014. `http://www.rfc-editor.org/rfc/rfc7235.txt`, `http://www.rfc-editor.org/rfc/rfc7235.txt`. 7

[6] Kitts, Brendan, Jing Ying Zhang, Gang Wu, Wesley Brandi, Julien Beasley, Kieran Morrill, John Ettedgui, Sid Siddhartha, Hong Yuan, Feng Gao *et al.*: *Click fraud detection: adversarial pattern recognition over 5 years at microsoft*. Em *Real World Data Mining Applications*, páginas 181–201. Springer, 2015. 8, 11, 12, 22, 35

[7] Daswani, Neil, Michael Stoppelman, Google Click Quality Team e Google Security Team: *The anatomy of clickbot.a*. 2007. 9, 10

[8] Leyden, John: *Botnet implicated in click fraud scam*. 2006. `https://www.theregister.co.uk/2006/05/15/google_adword_scam/`, acesso em 2017-09-20. 9, 10

[9] Court, United States District: *Microsoft vs eric lam et. al.* 2009. 9

[10] Pearce, Paul, Chris Grier, Vern Paxson, Vacha Dave, Damon McCoy, Geoffrey M. Voelker e Stefan Savage: *The zeroaccess auto-clicking and search-hijacking click fraud modules*. dezembro 2009. 9

[11] Thorpe, Simon, Denis Fize e Catherine Marlot: *Speed of processing in the human visual system*. nature, 381(6582):520, 1996. 15

[12] *Average reaction time*. `http://censusatschool.ca/data-results/2016-2017/average-reaction-time/`. 15

[13] Priebe, Jason: *A study of internet users' cookie and javascript settings*. abril 2009. http://www.smorgasbork.com/2009/04/29/a-study-of-internet-users-cookie-and-javascript-settings/, acesso em 10-02-2018. 16

[14] Winnicki, Andrzej: *Just how many web users really disable cookies or javascript?* abril 2016. https://blog.yell.com/2016/04/just-many-web-users-disable-cookies-javascript/, acesso em 10-02-2018. 16

[15] Miller, Brad, Paul Pearce, Chris Grier, Christian Kreibich e Vern Paxson: *What's clicking what? techniques and innovations of today's clickbots*. Em *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, páginas 164–183. Springer, 2011.

[16] Juels, Ari, Sid Stamm e Markus Jakobsson: *Combating click fraud via premium clicks.* Em *USENIX Security Symposium*, páginas 17–26, 2007.

[17] Daswani, Neil, Chris Mysen, Vinay Rao, Stephen Weis, Kourosh Gharachorloo e Shuman Ghosemajumder: *Online advertising fraud*. Crimeware: understanding new attacks and defenses, 40(2):1–28, 2008.

[18] Parsons, James: *The difference between website impressions and clicks*. janeiro 2015. https://growtraffic.com/blog/2015/01/difference-website-impressions-clicks, acesso em 25-09-2017.

[19] Willner, Barry E, Edith H Stern, Patrick J O'sullivan, Robert C Weir e Sean Callanan: *Cursor path vector analysis for detecting click fraud*, janeiro 2015. US Patent 8,938,395.

[20] O'sullivan, Patrick, Edith H Stern, Robert C Weir e Barry E Willner: *Pixel cluster transit monitoring for detecting click fraud*, fevereiro 2016. US Patent 9,251,522.

[21] Abraham, Ajith: *Rule-based expert systems*. Handbook of measuring system design, 2005.

[22] McMahan, H. Brendan, Gary Holt, D. Sculley, Michael Young, Dietmar Ebner, Julian Grady, Lan Nie, Todd Phillips, Eugene Davydov, Daniel Golovin, Sharat Chikkerur, Dan Liu, Martin Wattenberg, Arnar Mar Hrafnkelsson, Tom Boulos e Jeremy Kubica: *Ad click prediction: a view from the trenches*. 2013.

[23] P., Avila Clemenshia e Vijaya M. S.: *Click through rate prediction for display advertisement*. International Journal of Computer Applications, 136, fevereiro 2016.

[24] Dave, Kushal e Vasudeva Varma: *Predicting the click-through rate for rare/new ads*. abril 2010.

Apêndice A

Códigos do Publicador

Publisher.py

```
import http.server as http
from http.server import HTTPServer, BaseHTTPRequestHandler
from optparse import OptionParser
from os import curdir, sep

class RequestHandler(BaseHTTPRequestHandler):
    def handle_one_request(self):
        print(self.client_address[0])
        return http.BaseHTTPRequestHandler.handle_one_request(self)

    def do_GET(self):

        if self.path == "/":
            self.path = "/PublisherIndex.html"

        try:
            # Set MIME type
            sendReply = False
            if self.path.endswith(".html"):
                mimetype = 'text/html'
                sendReply = True
            if self.path.endswith(".jpg"):
                mimetype = 'image/jpg'
                sendReply = True
            if self.path.endswith(".gif"):
                mimetype = 'image/gif'
                sendReply = True
            if self.path.endswith(".js"):
                mimetype = 'application/javascript'
```

```
                sendReply = True
            if self.path.endswith(".css"):
                mimetype = 'text/css'
                sendReply = True
        except IOError:
            self.send_error(404, 'File Not Found: %s' % self.path)

        if sendReply:
            # Open the static file requested and send it
            try:
                with open(curdir + sep + self.path, 'rb') as f:
                    self.send_response(200)
                    self.send_header('Content-type', mimetype)
                    self.send_header("Set-Cookie", "foo=bar")
                    self.end_headers()
                    self.wfile.write(f.read())
            except FileNotFoundError:
                self.send_error(404, 'File Not Found: %s' % self.path)

    # def do_POST()
    # def do_PUT()
    # def do_DELETE()


def main():
    port = 8880
    print('Listening on localhost:%s' % port)
    server = HTTPServer(('', port), RequestHandler)
    server.serve_forever()

main()
```

PublisherIndex.html

```
<HTML>
    <HEAD>
        <TITLE>Publisher</TITLE>
    </HEAD>

    <BODY BGCOLOR="BBAADD">
        <HR>

        <a href="http://localhost:8881/">Link</a> for the ad network site.
```

```
        <P></P>
        <a id="adLink" href="javascript:void(0)">
            <img id="adImage" src="javascript:void(0)" width="320" height="180">
        </a>
        <script src="http://localhost:8881/adDisplayer.js"></script>

        <H1>Publisher</H1>

        <P> This is the Publisher site.

        <HR>
    </BODY>
</HTML>
```

Apêndice B

Códigos do Anunciante

Announcer.py

```
import http.server as http
from http.server import HTTPServer, BaseHTTPRequestHandler
from optparse import OptionParser
from os import curdir, sep


class RequestHandler(BaseHTTPRequestHandler):

    def handle_one_request(self):
        print(self.client_address[0])
        return http.BaseHTTPRequestHandler.handle_one_request(self)

    def do_GET(self):

        if self.path == "/":
            self.path = "/AnnouncerIndex.html"

        try:
            # Set MIME type
            sendReply = False
            if self.path.endswith(".html"):
                mimetype = 'text/html'
                sendReply = True
            if self.path.endswith(".jpg"):
                mimetype = 'image/jpg'
                sendReply = True
            if self.path.endswith(".gif"):
                mimetype = 'image/gif'
                sendReply = True
```

```
            if self.path.endswith(".js"):
                mimetype = 'application/javascript'
                sendReply = True
            if self.path.endswith(".css"):
                mimetype = 'text/css'
                sendReply = True
        except IOError:
            self.send_error(404, 'File Not Found: %s' % self.path)


        if sendReply:
            # Open the static file requested and send it
            try:
                with open(curdir + sep + self.path, 'rb') as f:
                    self.send_response(200)
                    self.send_header('Content-type', mimetype)
                    self.send_header("Set-Cookie", "foo=bar")
                    self.end_headers()
                    self.wfile.write(f.read())
            except FileNotFoundError:
                self.send_error(404, 'File Not Found: %s' % self.path)

    #def do_POST()
    #def do_PUT()
    #def do_DELETE()


def main():
    port = 8882
    print('Listening on localhost:%s' % port)
    server = HTTPServer(('', port), RequestHandler)
    server.serve_forever()

main()
```

AnnouncerPage.html

```
<HTML>
    <HEAD>
        <TITLE>Announcer</TITLE>
    </HEAD>

    <BODY BGCOLOR="DDBBAA">
        <HR>
```

```
        <a href="http://localhost:8880/">Link</a> back to Publisher.

        <H1>Announcer</H1>

        <P> This is the Announcer site.

        <HR>
    </BODY>
</HTML>
```

Apêndice C

Códigos Python da Rede de Anúncios

Server.py

```
import http.server as http
from http.server import HTTPServer, BaseHTTPRequestHandler
from os import curdir, sep
from jinja2 import Template
from socketserver import ThreadingMixIn
import time

from onlineDefense import *
from offlineDefense import *
from Persistence import *
from Hasher import *


class RequestDeque():
    def __init__(self, length):
        self.reqs = deque(maxlen=length)
        self.lock = threading.Lock()

    def getDeque(self):
        self.lock.acquire()
        try:
            if self.reqs:
                reqs = self.reqs
            else:
                reqs = []
        finally:
            self.lock.release()
        return reqs

    def append(self, req):
        self.lock.acquire()
        try:
            self.reqs.append(req)
        finally:
```

```
            self.lock.release()


class TempRequest():
    def __init__(self, req, reports, status):
        self.req = req
        self.reports = reports
        self.status = status


class LoaderRequests():
    def __init__(self):
        self.lock = threading.RLock()
        self.tempReqs = []
        self.delta = 3.0

    def insert(self, req, reports, status):
        self.lock.acquire()
        try:
            self.tempReqs.append(TempRequest(req, reports, status))
            count = 0
            for treq in self.tempReqs:
                req = treq.req
                print(bcolors.bluecyan + str(req) + bcolors.end)
                print()
                count += 1
                if count > 5:
                    break
            print(len(self.tempReqs))
        finally:
            self.lock.release()

    def fetch(self, newReq, delta=3.0):
        delta = datetime.timedelta(seconds=delta)
        prevReq = None
        prevReports = None
        prevStatus = None
        self.lock.acquire()
        try:
            for treq in self.tempReqs:
                req = treq.req
                print('\033[1;34;43m' + str(newReq.ip) + ' == ' + str(req.ip))
                print(str(newReq.user_agent) + ' == ' + str(req.user_agent))
                print(str(newReq.date) + ' - ' + str(req.date) + ' = ' +
                      str(newReq.date - req.date) + ' <= ' + str(delta))
                if req.ip == newReq.ip and req.user_agent == newReq.user_agent:
                    if newReq.date - req.date <= delta:
                        print('Match found!')
                        prevReq = req
                        prevReports = treq.reports
                        prevStatus = treq.status
                        self.tempReqs.remove(treq)
                        break
                    else:
                        print('Too long since previous request.')
                else:
                    print('Different.')
```

```
            finally:
                print('\033[0m')
                self.lock.release()
            return prevReq, prevReports, prevStatus

        def show(self):
            self.lock.acquire()
            print(bcolors.magcyan + '%%%%%%%%%%')
            try:
                for treq in self.tempReqs:
                    print(treq.req)
                    for report in treq.reports:
                        print(report)
                    print(treq.status)
                    print('      <00000000000> ')
            finally:
                print('%%%%%%%%%%\n' + bcolors.end)
                self.lock.release()


class RequestHandler(BaseHTTPRequestHandler):

    MAX_REQUESTS = 10
    requestDeque = RequestDeque(MAX_REQUESTS)
    adReqAnalyzer = AdRequestAnalyzer()
    onlineAnalyzer = OnlineClickAnalyzer()
    loaderRequests = LoaderRequests()

    def handle_one_request(self):
        #print(self.client_address[0])
        return http.BaseHTTPRequestHandler.handle_one_request(self)

    def do_GET(self, redirection=False):
        print('\n\n----------> GET START ----------')
        print('//////////////////////////////////////////////////\n')
        self.path = correctHash(self.path)

        print(bcolors.whiteblue + str(self.headers) + bcolors.end)

        req = ClickRequest(self.client_address[0], dt.now(),
                           self.path, self.headers)
        reqType = ''

        print(bcolors.blackgreen + req.path + bcolors.end)

        lock = threading.Lock()
        lock.acquire()
        try:
            self.path, reqType = self.__correctPath(req)
            print('>>>' + str(self.path) + '<<<')
            # Changes self.path accordingly
        except Exception as exc:
            print(exc)
        finally:
            lock.release()

        try:
```

```
            mimetype, sendReply = self.__determineType(self.path)
        except IOError:
            self.send_error(404, 'File Not Found: %s' % self.path)
            sendReply = True

        print(req)

        urlEnd = ''
        if self.isPageRequest(mimetype):
            urlEnd = urlHash(req, 'localhost:8882')

        if reqType == 'adRequest':
            clickStatus = self.__adRequestProcedure(req)
        elif reqType == 'redirect':
            clickStatus = self.__redirectProcedure(req)
        else:
            print('??? ' + reqType + ' ???')
            clickStatus = self.__miscProcedure(req)

        if sendReply:
            # Open the static file requested and send it
            if self.isPageRequest(mimetype):
                self.pageResponse(req, mimetype, reqType, urlEnd)
            elif self.isImageRequest(mimetype):
                self.imageResponse(mimetype)
        print('\n//////////////////////////////////////////////////')
        print('<---------- GET END <----------\n\n')

    def __adRequestProcedure(self, req):
        print('!!! adRequest !!!')
        # Test click with adRequestAnalyzer and guard request + result
        # into TempRequest object, then guard into loaderRequests object.
        clickStatus, reports = self.adReqAnalyzer.analyze(req, self.requestDeque)
        self.requestDeque.append(req)
        self.loaderRequests.insert(req=req, reports=reports, status=clickStatus)
        return clickStatus

    def __redirectProcedure(self, req, verbose=False):
        print('!!! redirect !!!')
        # There's a request for this page, so there should be a similar
        # request for the adRequest page. Find it.
        prevReq, prevReports, prevStatus = self.loaderRequests.fetch(newReq=req)

        if verbose:
            print('\t$$$$$$$$$$$$$$$ start $$$$$$$$$$$$$$$$$')
            print(prevReq)
            print(prevStatus)
            for report in prevReports:
                print(report)
            print('\t$ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $')
            print(req)
            print('\t$$$$$$$$$$$$$$$ end $$$$$$$$$$$$$$$$$')

        if (prevStatus == ClickStatus.Fraud) or (prevReq is None):
            # If the click failed on the previous tests, it's considered a fraud.
            # If the corresponding request wasn't found, something's wrong!
            clickStatus = ClickStatus.Fraud
```

```
            reports = []
        else:
            # Analyze the redirect.html request as well.
            clickStatus, reports = self.onlineAnalyzer.analyze(
                req, prevDate=prevReq.date)
            if clickStatus == ClickStatus.Fraud:
                prevStatus = clickStatus

        # Log both clicks to the database
        if prevReq is not None:
            prevReportId = self.__logClick(prevReq, prevReports, prevStatus)
            for report in prevReports:
                reports.append(report)
            self.__logClick(req, reports, clickStatus, prevReportId)
        else:
            self.__logClick(req, reports, clickStatus)

        return clickStatus

    def __miscProcedure(self, req):
        # clickStatus = self.onlineAnalyzer.analyze(req, self.requestDeque)
        # self.requestDeque.append(req)
        status = ClickStatus.Valid
        self.__logNormalRequest(req, status)

        return status

    def pageResponse(self, req, mimetype, reqType, urlEnd = ''):
        if self.path.startswith('http://localhost:8881/adRequest/'):
            self.path = "index_template.html"

        with open(curdir + sep + self.path, 'rb') as f:
            page = ''
            if self.path.endswith('adDisplayer.js'):
                for line in f:
                    line = line.decode()
                    if line.find('@AD_LINK@') > -1:
                        url = 'http://localhost:8881/adRequest/' + str(urlEnd)
                        line = line.replace('@AD_LINK@', url)
                    page = ''.join([page, line])
            else:
                for line in f:
                    page = ''.join([page, line.decode()])

            print(page)
            t = Template(page)
            print('----')
            rend = t.render(something=req.ip)
            print('>' + str(type(rend)) + '<')
            print(rend)

            #if self.path.endswith('')
            #if reqType == 'adRequest':
                #self.send_response(302)
                #self.send_header('Location', 'http://localhost:8881/redirect.html')
                #self.send_header('Set-Cookie', 'jsEnabled=true')
            #elif reqType == 'redirect':
```

```
            #self.send_response(302)
            #self.send_header('Location', 'http://localhost:8882/')
        #else:
        if reqType == '404':
            self.send_response(404)
        else:
            self.send_response(200)
            self.send_header('Content-type', mimetype)
        self.end_headers()
        self.wfile.write(rend.encode())
        print('Response sent!')

def imageResponse(self, mimetype):
    try:
        urlParts = self.path.split('/')
        img = urlParts[-1]
        with open(curdir + sep + img, 'rb') as f:
            print(curdir+sep+img)
            self.send_response(200)
            self.send_header('Content-type', mimetype)
            self.end_headers()
            self.wfile.write(f.read())
            print('Response sent!')
    except Exception as exc:
        print(exc)

def isImageRequest(self, mimetype):
    return mimetype == 'image/png' or mimetype == 'image/jpg' or\
           mimetype == 'image/gif' or mimetype == 'image/x-icon'

def isPageRequest(self, mimetype):
    return mimetype == 'text/html' or mimetype == 'application/javascript'\
           or mimetype == 'text/css'

def __determineType(self, path):
    mimetype = ''
    sendReply = False
    if path.endswith(".html"):
        mimetype = 'text/html'
        sendReply = True
    if path.endswith(".jpg"):
        mimetype = 'image/jpg'
        sendReply = True
    if path.endswith(".png"):
        mimetype = 'image/png'
        sendReply = True
    if path.endswith(".gif"):
        mimetype = 'image/gif'
        sendReply = True
    if path.endswith(".js"):
        mimetype = 'application/javascript'
        sendReply = True
    if path.endswith(".css"):
        mimetype = 'text/css'
        sendReply = True
    if path.endswith(".ico"):
        mimetype = 'image/x-icon'
```

```
            sendReply = True
        return mimetype, sendReply

    def __correctPath(self, req):
        reqType = ''
        if self.path == "/":
            return "/404.html", '404'
        elif self.path.startswith('/adRequest/'):
            # This request should be the "click" itself.
            return self.__compareUrlHash(req), 'adRequest'
        else:
            type = self.path.split('/')[-1]
            type = type.split('.')[0]
            return self.path, type

    def __compareUrlHash(self, req):
        receivedHash = self.path.split('/', 2)[2]
        currentHash = urlHash(req, 'localhost:8882')

        print('>>>>>>>>>>>>>>>>>>>>>>>>>>>>>')
        print(str(threading.current_thread()) + ' received: ' + receivedHash)
        print(str(threading.current_thread()) + ' current: ' + currentHash)
        print('<<<<<<<<<<<<<<<<<<<<<<<<<<<<<')

        if currentHash == receivedHash:
            return "/index_template.html"
        else:
            return "/404.html"

    def __logNormalRequest(self, req, clickStatus):
        logStr = "dbname=postgres host=localhost port=5432 " \
                 "user=postgres password=postgres"
        dbLog = DatabaseLogger(logStr)
        dbLog.logRequest(req, clickStatus)

    def __logClick(self, req, reports, clickStatus, prevReportId=None):
        logStr = "dbname=postgres host=localhost port=5432 " \
                 "user=postgres password=postgres"
        dbLog = DatabaseLogger(logStr)
        return dbLog.logClick(req, reports, clickStatus, prevReportId)


    #def do_POST():

    #def do_PUT():

    #def do_DELETE():

    #def do_HEAD():


class ThreadedHTTPServer(ThreadingMixIn, HTTPServer):
    """Handle requests in a separate thread."""


def tprint(name, str):
    print('(' + name + '): ' + str)
```

```
def main():
    isOnline = getHandlerConfig()
    # isOnline == 1 --> server + online analysis (default)
    # isOnline == -1 --> offline analysis
    # isOnline == 0 --> both options
    offHandler = OfflineHandler()
    try:
        if isOnline==-1 or isOnline==0:
            threading.Thread(target=offHandler.handle, args=()).start()
        elif isOnline==1 or isOnline==0:
            threading.Thread(target=startServer, args=()).start()
    except Exception as exc:
        print(exc)


def getHandlerConfig(filename='handlerConfig.txt'):
    value = 1
    with open(filename, 'r') as f:
        for line in f:
            if line.startswith('isOnline'):
                value = int(line.split('=')[1])
    return value


def startServer():
    port = 8881
    print('Listening on localhost:%s' % port)
    server = ThreadedHTTPServer(('', port), RequestHandler)
    server.serve_forever()

if __name__ == "__main__":
    main()
```

Request.py

```
import datetime
from enum import Enum


class bcolors:
# General use
    end = '\033[0m'
    bold = '\033[1m'

# Rules
    redbg = '\033[1;31;40m'
    greenbg = '\033[1;32;40m'
    yellowbg = '\033[1;33;40m'
    bluebg = '\033[1;34;40m'
    magbg = '\033[1;35;40m'
    cyanbg = '\033[1;36;40m'
    whitebg = '\033[1;37;40m'
```

```
# Debug
    debug1 = '\033[1;37;41m'  # cyan + white
    debug2 = '\033[1;36;41m'  # cyan + red

# Offline Text
    offline1 = '\033[1;30;42m'  # black + green
    offline2 = '\033[1;31;42m'  # red + green
    offline3 = '\033[1;34;42m'  # blue + green
    offline4 = '\033[1;35;42m'  # mag + green

# Assorted
    blackcyan = '\033[1;30;46m'
    redcyan = '\033[1;31;46m'
    bluecyan = '\033[1;34;46m'
    magcyan = '\033[1;35;46m'
    whiteblue = '\033[1;37;44m'
    blackgreen = '\033[1;30;42m'



class ClickRequest():
    def __init__(self, ip, date, path, headers):
        self.ip = ip
        self.date = date
        self.path = path
        self.headers = headers
        self.user_agent = ''
        self.accept = '*/*'
        self.accept_lang = ''
        self.accept_encode = ''
        self.referer = ''
        self.dnt = -1
        self.connection = ''
        self.cookiesStr = ''
        self.cookies = []
        # Set the parameters using self.headers
        self.__initHeaders(self.headers.as_string())

    def __str__(self):
        string = '\n------ Request Start ------>\n'
        string = ''.join([string, "Address: ", str(self.ip), "\n"])
        string = ''.join([string, "Date: ", str(self.date), "\n"])
        string = ''.join([string, "Request path: ", str(self.path), '\n'])
        string = ''.join([string, "Request headers: ", str(self.headers), '\n'])
        for cookie in self.cookies:
            string = ''.join([string, '<',cookie.getCookie(), '>; '])
        string = ''.join([string, '\n<------ Request End ------\n'])
        return string

    def __initHeaders(self, headers):
        # The method ignores cookies with name/value pair of 'foo=bar'.
        # This could be changed to apply to other cookies.
        for line in headers.split('\n'):
            print(line)
            if line.find('User-Agent: ') > -1:
                self.user_agent = line.split(': ')[1]
```

```
            if line.find('Accept: ') > -1:
                self.accept = line.split(': ')[1]
            if line.find('Accept-Language: ') > -1:
                self.accept_lang = line.split(': ')[1]
            if line.find('Accept-Encoding: ') > -1:
                self.accept_encode = line.split(': ')[1]
            if line.find('Referer: ') > -1:
                self.referer = line.split(': ')[1]
            if line.find('DNT: ') > -1:
                self.dnt = int(line.split(': ')[1])
            if line.find('Connection: ') > -1:
                self.connection = line.split(': ')[1]
            if line.find('Cookie: ') > -1:
                self.cookiesStr = line.split(': ')[1]
                self.__clearCookies()
                cookies = line.split(': ')[1].split(';')
                for cookie in cookies:
                    cookie = cookie.strip()
                    if cookie != 'foo=bar' and cookie != '' and cookie is not None:
                        self.cookies.append(RequestCookie(cookie.strip()))

    def writeToFile(self, filename):
        if filename is not None:
            with open(filename, 'w') as f:
                f.write("\n----- Request Start ----->\n")
                f.write("Address: " + str(self.ip) + "\n")
                f.write("Date: " + str(self.date) + "\n")
                f.write("Request path: " + str(self.path) + '\n')
                f.write("Request headers: " + str(self.headers) + '\n')
                f.write("<----- Request End -----\n")

    def getHeaderValue(self, header):
        headerList = self.headers.as_string().split('\n')
        for headerLine in headerList:
            if headerLine.find(header) > -1:
                pos = headerLine.find(': ')
                return headerLine[pos+2:]
        return ''

    def __clearCookies(self):
        # Function ignores certain cookies. As implemented, only 'foo=bar'
        self.cookiesStr = self.cookiesStr.replace('; foo=bar', '')
        self.cookiesStr = self.cookiesStr.replace('foo=bar; ', '')
        self.cookiesStr = self.cookiesStr.replace('foo=bar', '')


class RequestCookie():
    def __init__(self, cookie):
        self.__parseCookie(cookie)

    def __parseCookie(self, cookie):
        self.name, self.value = cookie.split('=')

    def getCookie(self):
        return str(self.name) + '=' + str(self.value)
```

```
class ClickStatus(Enum):
    Valid = 1
    Fraud = 0
```

Hasher.py

```
import hashlib


def isInUrlRange(byte):
    return byte in range(45, 47) or \
           byte in range(48, 57) or \
           byte in range(65, 90) or \
           byte in range(97, 122) or \
           byte == 126


def urlHash(req, announcerAddress, verbose=False):
    m = hashlib.sha256()

    referer = req.getHeaderValue('Referer')
    user_agent = req.getHeaderValue('User-Agent')
    accept = req.getHeaderValue('Accept')

    string = ''.join([str(req.ip), str(user_agent),
                      str(referer), str(announcerAddress)])
    m.update(string.encode('utf-8'))
    result = m.digest()
    hexd = m.hexdigest()

    urlString = 'h'
    for i in range(len(result)):
        byte = result[i]
        if isInUrlRange(byte):
            urlString = ''.join([urlString, chr(byte)])
        else:
            hexByte = hex(byte).upper()
            urlString = ''.join([urlString, '%', hexByte[2:]])

    if verbose:
        print('   ! ! ! <' + referer + '>, <' + user_agent + '>, <' +
              accept + '> ! ! !')
        print('   ! ! ! <' + user_agent + '> ! ! !')
        print('@ @ @ Start @ @ @')
        print(type(result))
        print(result)
        print(hexd)
        print(urlString)
        print('@ @ @ End @ @ @')

    return urlString


def correctHash(path):
```

```
    # '%25' is the code for the '%' character
    if path.find('%25') > -1:
        return path.replace('%25', '%')
    return path
```

Rules.py

```
from Request import *
from Persistence import DatabaseLogger
from collections import deque


class UserBehavior():
    def __init__(self, behavior=None):
        if behavior:
            self.load(behavior)

    def load(self, behavior):
        self.id = behavior[0]
        self.date = behavior[1]
        self.clicks_first = behavior[2]
        self.clicks_other = behavior[3]
        self.scrolls_first = behavior[4]
        self.scrolls_other = behavior[5]
        self.events_first = behavior[6]
        self.events_other = behavior[7]
        self.time_first = behavior[8]
        self.time_other = behavior[9]
        self.visited = behavior[10]


class RequestFromDB():
    def __init__(self, row):
        self.id = row[0]
        self.ip_address = row[1]
        self.date = row[2]
        self.path = row[3]
        self.referer = row[4]
        self.user_agent = row[5]
        self.accept = row[6]
        self.acc_encode = row[7]
        self.acc_lang = row[8]
        self.dnt = row[9]
        self.cookies = row[10]
        if row[11] == 'valid':
            self.status = ClickStatus.Valid
        elif row[11] == 'fraud':
            self.status = ClickStatus.Fraud
        else:
            self.status = None
        self.report_id = row[12]

    def __str__(self):
        string = '\n'.join([str(self.id), str(self.ip_address), str(self.date),
```

```
                                str(self.path), str(self.referer),
                                str(self.user_agent), str(self.accept),
                                str(self.acc_encode), str(self.acc_lang),
                                str(self.dnt), str(self.cookies),
                                str(self.status), str(self.report_id)])
        return string


#--------------------
#------ RULES -------
#--------------------


class ClickRule():
    def __init__(self, weight=0):
        self.name = ''
        self.weight = weight

    def testClick(self, req):
        return ClickStatus.Valid


class DecisiveRule(ClickRule):
    def __init__(self, weight=0):
        super().__init__(weight)

    def testClick(self, req):
        super().testClick(req)


class IndicativeRule(ClickRule):
    def __init__(self, weight=0):
        super().__init__(weight)

    def testClick(self, req):
        super().testClick(req)


# -----------------
# --- OFFLINE RULES

class PagesLoadedRule(DecisiveRule):
    def __init__(self, weight=1):
        super().__init__(weight)

    def testClick(self, req, newReqs):
        # Expected pages: adDisplayer.js, announcerAd.png,
        #   adRequest, shouldLoad.png, redirect, favicon.ico
        # Expected not to load: hidden.png

        status = ClickStatus.Valid
        dbLogger = DatabaseLogger('dbname=postgres host=localhost port=5432 '
                                  'user=postgres password=postgres')
        print(bcolors.cyanbg +
              ' %%%%%%%% PAGES LOADED RULE %%%%%%%% \n')

        query = 'select distinct on (path) id, req_date, path from requests ' \
```

```
        'where path = \'/adDisplayer.js\' ' \
        'or path = \'/announcerAd.png\' and req_date < \'' + \
        str(req.date) + '\' order by path, req_date desc;'
rows = dbLogger.genericFetch(query)

# Could have been loaded before
adDisplayerOK = False
announcerAdOK = False
for row in rows:
    print(row[2])
    if row[2] == '/announcerAd.png':
        print('/announcerAd.png OK')
        announcerAdOK = True
    elif row[2] == '/adDisplayer.js':
        print('/adDisplayer.js OK')
        adDisplayerOK = True

query = self.__makeQuery(req.ip_address, req.date)
rows = dbLogger.genericFetch(query)

# Must have been loaded within a short time
adRequestOK = False
shouldLoadOK = False
redirectOK = False
faviconOK = False
hiddenOK = False
for row in rows:
    print(row[2])
    if row[2] == '/adRequest.html':
        print('/adDisplayer.js OK')
        adRequestOK = True
    elif row[2] == '/shouldLoad.html':
        print('/adDisplayer.js OK')
        shouldLoadOK = True
    elif row[2] == '/redirect.html':
        print('/adDisplayer.js OK')
        redirectOK = True
    elif row[2] == '/hidden.png':
        print('/adDisplayer.js OK')
        hiddenOK = True
    elif row[2] == '/favicon.ico':
        print('/favicon.ico OK')
        faviconOK = True

if (adDisplayerOK and announcerAdOK and adRequestOK and
        shouldLoadOK and redirectOK and faviconOK) \
        and not hiddenOK:
    print(' ==> PASSED')
    status = ClickStatus.Valid
else:
    print(' ==> FAILED')
    status = ClickStatus.Fraud

print('\n %%%%%%%% PAGES LOADED RULE END %%%%%%%% \n'
      + bcolors.end + '\n')
return status
```

```
    def __makeQuery(self, ip, date, delta=datetime.timedelta(seconds=5)):
        query = 'select distinct on (path) id, req_date, path, ' \
                'report_id, status from requests where '

        startDate = ''.join(['timestamp \'', str(date),
                             '\' - interval \'', str(delta), '\''])
        endDate = ''.join(['timestamp \'', str(date),
                           '\' + interval \'', str(delta), '\''])
        conditions = 'ip_address = \'' + str(ip) + \
                     '\' and req_date between (' + startDate + \
                     ') and (' + endDate + ') '

        order = 'order by path, req_date desc;'

        query = ''.join([query, conditions, order])
        return query



class TimePeriodRule(IndicativeRule):
    def __init__(self, weight=1):
        super().__init__(weight)

    def testClick(self, req,
                  shortPeriod = datetime.timedelta(seconds=30),
                  consistencyPeriod = datetime.timedelta(minutes=10),
                  consistencyLeeway = datetime.timedelta(seconds=40),
                  verbose=True):

        print(bcolors.whitebg +
              ' %%%%%%%% TIME PERIOD RULE %%%%%%%% \n')
        status = ClickStatus.Valid

        dbLogger = DatabaseLogger('dbname=postgres host=localhost port=5432 '
                                  'user=postgres password=postgres')
        query = self.__makeQuery(req.ip_address, req.user_agent,
                                 req.date, consistencyPeriod)
        print(query)
        print()
        rows = dbLogger.genericFetch(query)
        size = len(rows)

        reqPos = 0
        print(req)
        for i, row in enumerate(rows):
            if row[0] == req.id:
                print('>>>> ' + str(i) + ' <<<<')
                reqPos = i
                break

        for row in rows:
            print(row[0])

        if reqPos >= 0 and size >= 3:
            # Verify if there a chain of requests that fit within
            #  the test's failing conditions
            for i in range(reqPos-4, reqPos+1):
```

```
            if i >= 0:
                # Check for 3 consecutive requests in within shortPeriod
                if i >= reqPos-2 and i+2 < size:
                    print('three: ' + str(rows[i][1]) +
                          '\n\t and' + str(rows[i + 2][1]))
                    if rows[i + 2][1] - rows[i][1] <= shortPeriod:
                        print('3 - FRAUD')
                        status = ClickStatus.Fraud
                        break

                # Check for 5 requests within the same average
                #  leeway within consistencyPeriod
                if i >= reqPos-4 and i+4 < size:
                    print('five: ' + str(rows[i]) + '\n\t and' +
                          str(rows[i + 4]))
                    if rows[i + 4][1] - rows[i][1] <= consistencyPeriod:
                        if not self.__testAvgTime(rows, i, i + 4,
                                                  consistencyLeeway):
                            print('5 - FRAUD')
                            status = ClickStatus.Fraud
                            break

    if status == ClickStatus.Fraud:
        print(' ==> FAILED')
    else:
        print(' ==> PASSED')
    print('\n %%%%%%%% TIME PERIOD RULE END %%%%%%%% \n' +
          bcolors.end + '\n')
    return status


def __testAvgTime(self, rows, posStart, posEnd, leeway):
    prevDatetime = None
    deltas = []
    print('\n')
    for i in range(posStart, posEnd+1):
        currDatetime = rows[i][1]
        if prevDatetime is not None:
            print(str(currDatetime) + ' - ' + str(prevDatetime) + ' = ')
            print(currDatetime - prevDatetime)
            deltas.append(currDatetime - prevDatetime)
        prevDatetime = currDatetime

    avgTime = sum(deltas, datetime.timedelta(0)) / len(deltas)
    print('\\\\\\\\\\')
    print(avgTime)
    print(leeway)
    print('\\\\\\\\\\')

    for i in range(len(deltas)):
        print(deltas[i])
        if abs(deltas[i] - avgTime) >= leeway:
            print('true\n')
            return True
    print('false\n')
    return False
```

```
    def __makeQuery(self, ip, agent, date, maxTime):
        query = 'select id, req_date, report_id, status from requests where '

        path = '\'/adRequest.html\''
        restrictions = ''.join(['ip_address = \'', str(ip),
                                '\' and\n user_agent = \'', agent,
                                '\' and\n path = ', path, ' and\n'])

        startDate = ''.join(['timestamp \'', str(date), '\' - interval \'',
                             str(maxTime), '\''])
        endDate = ''.join(['timestamp \'', str(date), '\' + interval \'',
                           str(maxTime), '\''])
        dateStr = ''.join(['req_date between (', startDate, ') and (',
                           endDate, ')'])

        query = ''.join([query, restrictions, dateStr, ' order by req_date;'])
        return query


class BehaviorRule(IndicativeRule):
    def __init__(self, weight=1):
        super().__init__(weight)

    def testClick(self, reqDB):
        # uses DBLogger to fetch current blacklist
        # if req.ip is in the list, the click fails this test
        status = ClickStatus.Valid
        dbLogger = DatabaseLogger('dbname=postgres host=localhost port=5432'
                                  ' user=postgres password=postgres')
        '''WHERE session_date < '$date_string'
            ORDER BY session_date DESC
            LIMIT 1;'''
        query = 'select * from behavior where ip_address = \'' + \
                str(reqDB.ip_address) + '\' and access_date < \'' + \
                str(reqDB.date) + '\' order by access_date desc ' + 'limit 1;'
        row = dbLogger.genericFetch(query)
        print(row)
        print(bcolors.yellowbg +
              ' %%%%%%%% BEHAVIOR RULE %%%%%%%% \n')
        if row:
            behavior = UserBehavior(behavior=row)
            if self.__behaviorTest(behavior):
                print(' ==> PASSED')
            else:
                print(' ==> FAILED')
            print('\n %%%%%%%% BEHAVIOR RULE END %%%%%%%% \n' +
                  bcolors.end + '\n')
            return status
        else:
            print(' ==> NO RESULT')
            print('\n %%%%%%%% BEHAVIOR RULE END %%%%%%%% \n' +
                  bcolors.end + '\n')
            return None

    def __behaviorTest(self, bhv):
        # TODO
```

```
        result = True
        if bhv.clicks_first > 0:
            # User went to other pages, most likely
            if bhv.time_first < datetime.timedelta(seconds=3):
                result = False
        else:  # clicks_first == 0
            # User can't have gone to other pages
            if bhv.scrolls_first < 1:
                result = False

        result = (bhv.clicks_first > 0 and bhv.clicks_other > 0) or \
                 (bhv.scrolls_first > 1 and bhv.scrolls_other > 1) or \
                 (bhv.events_first > 2 and bhv.events_other > 2) or \
                 (bhv.time_first > datetime.timedelta(seconds=10) and
                  bhv.time_other > datetime.timedelta(minutes=2)) or \
                 (bhv.visited > 1)
        return result


# ------------------
# --- ONLINE RULES
class RedirectTimeRule(IndicativeRule):
    def __init__(self, weight=1):
        super().__init__(weight)

    def testClick(self, req, prevDate=None):
        expected = datetime.timedelta(seconds=0.7)
        print(bcolors.cyanbg +
              ' %%%%%%%% REDIRECT TIME RULE %%%%%%%% \n')

        print(str(req.date) + ' - ' + str(prevDate) + ' = ' +
              str(req.date - prevDate) + ' > ' + str(expected))
        if req.date - prevDate < expected:
            print(' ==> PASSED')
            status = ClickStatus.Valid
        else:
            print(' ==> FAILED')
            status = ClickStatus.Fraud
        print('\n %%%%%%%% TIMER RULE END %%%%%%%% \n' +
              bcolors.end + '\n')
        return status



class JavascriptEnabledRule(IndicativeRule):
    def __init__(self, weight=1):
        super().__init__(weight)

    def testClick(self, req):
        print(bcolors.greenbg +
              ' %%%%%%%% JAVASCRIPT RULE %%%%%%%% \n')
        status = ClickStatus.Fraud
        for cookie in req.cookies:
            if cookie.name == 'JSEnabled' and \
               cookie.value == self.__createHash(req):
                status = ClickStatus.Valid
        if status == ClickStatus.Fraud:
```

```
            print(' ==> FAILED')
        else:
            print(' ==> PASSED')
        print('\n %%%%%%%% JAVASCRIPT RULE END %%%%%%%% \n' +
              bcolors.end + '\n')
        return status

    def __createHash(self, req):
        return 'true'



class AcceptLangRule(DecisiveRule):
    def __init__(self, weight=1):
        super().__init__(weight)

    def testClick(self, req):
        print(bcolors.bluebg +
              ' %%%%%%%% ACCEPT LANGUAGE RULE %%%%%%%% \n')
        if req.accept_lang.strip() == '':
            print(' ==> FAILED')
            status = ClickStatus.Fraud
        else:
            print(' ==> PASSED')
            status = ClickStatus.Valid
        print('\n %%%%%%%% ACCEPT LANGUAGE RULE END %%%%%%%% \n' +
              bcolors.end + '\n')
        return status



class DNTRule(IndicativeRule):
    def __init__(self, weight=1):
        super().__init__(weight)

    def testClick(self, req):
        print(bcolors.yellowbg + ' %%%%%%%% DNT RULE %%%%%%%% \n')
        if req.dnt > -1:
            print(' ==> PASSED')
            status = ClickStatus.Valid
        else:
            print(' ==> FAILED')
            status = ClickStatus.Fraud
        print('\n %%%%%%%% DNT RULE END %%%%%%%% \n' +
              bcolors.end + '\n')
        return status



class HumanTimerRule(DecisiveRule):
    def __init__(self, weight=1, delta=0.5):
        super().__init__(weight)
        self.delta = datetime.timedelta(seconds=delta)

    def testClick(self, req, requestDeque):
        reqDeque = deque(requestDeque.getDeque())
```

```
        status = ClickStatus.Valid
        print(bcolors.redbg + ' %%%%%%%% TIMER RULE %%%%%%%% \n')
        for prevReq in reqDeque:
            print(' ---->' + str(req.ip) + ' : ' + str(req.date))
            print(' ---->' + str(prevReq.ip) + ' : ' + str(prevReq.date))
            if req.ip == prevReq.ip:
                difference = req.date - prevReq.date
                print('\t' + str(difference) + ' vs ' + str(self.delta))
                print()
                if difference < self.delta:
                    status = ClickStatus.Fraud
                    break

        if status == ClickStatus.Fraud:
            print(' ==> FAILED')
        else:
            print(' ==> PASSED')
        print('\n %%%%%%%% TIMER RULE END %%%%%%%% \n' +
              bcolors.end + '\n')

        return status



class UserAgentRule(DecisiveRule):
    def __init__(self, weight=1):
        super().__init__(weight)

    def testClick(self, req):
        print(bcolors.magbg +
              ' %%%%%%%% USER AGENT RULE %%%%%%%% \n')
        fields = req.user_agent.split(' ')
        productList = []
        commentList = []

        i = 0
        max = len(fields)
        while i < max:
            field = fields[i]
            if field.startswith('('):
                product = field
                if not field.endswith(')'): # comment
                    product = field
                    i = i+1
                    product += (' ' + fields[i])
                    while not field.endswith(')'):
                        i = i + 1
                        field = fields[i]
                        product += (' ' + field)
                commentList.append(product)
            else: # product
                productList.append(field)

            i = i+1

        status = self.testUserAgent(productList, commentList)
        print('\n %%%%%%%% USER AGENT RULE END %%%%%%%% \n' +
```

```
            bcolors.end + '\n')
        return status

    def testUserAgent(self, prodList, commList):
        for comm in commList:
            print(comm)
        #   if not ... :
        #       return ClickStatus.Fraud

        for prod in prodList:
            print(prod)
            if not ( prod.startswith('Mozilla') or
                    prod.startswith('Chrome') or
                    prod.startswith('AppleWebKit') or
                    prod.startswith('Firefox') or
                    prod.startswith('Safari') or
                    prod.startswith('Opera') or
                    prod.startswith('Gecko') or
                    prod.startswith('InternetExplorer') ):
                print(' ==> FAILED')
                return ClickStatus.Fraud

        print(' ==> PASSED')

        return ClickStatus.Valid



class BlacklistRule(DecisiveRule):
    def __init__(self, weight=1):
        super().__init__(weight)

    def testClick(self, req):
        # uses DBLogger to fetch current blacklist
        # if req.ip is in the list, the click fails this test
        status = ClickStatus.Valid
        query = 'select ip_address from blacklist;'
        dbLogger = DatabaseLogger('dbname=postgres host=localhost port=5432'
                                  ' user=postgres password=postgres')
        blacklist = self.__getBlacklist(dbLogger.genericFetch(query))
        print(bcolors.whitebg +
              ' %%%%%%%% BLACKLIST RULE %%%%%%%% \n')
        if blacklist:
            for address in blacklist:
                if req.ip == address:
                    status = ClickStatus.Fraud
                    print(' ==> FAILED')
        if status == ClickStatus.Valid:
            print(' ==> PASSED')
        print('\n %%%%%%%% BLACKLIST RULE END %%%%%%%% \n' +
              bcolors.end + '\n')
        return status

    def __getBlacklist(self, rows):
        blacklist = []
        for row in rows:
            blacklist.append(row[0])
```

```
        return blacklist
```

OnlineDefense.py

```
import threading
from datetime import datetime as dt
import psycopg2

from rules import *


class RuleReport():
    def __init__(self, name=None, isDecisive=None, weight=None, result=None):
        self.name = name
        self.isDecisive = isDecisive
        self.weight = weight
        if result == ClickStatus.Fraud:
            self.result = False
        elif result == ClickStatus.Valid:
            self.result = True
        else:
            self.result = None
        self.write()

    def __str__(self):
        return self.name + ' // ' + str(self.isDecisive) + ' // ' + \
               str(self.weight) + ' // ' + str(self.result)

    def write(self):
        print(bcolors.whiteblue + self.__str__() + bcolors.end + '\n')


class ClickAnalyzer():

    def __init__(self):
        self.rules = []

    def analyze(self, requests=None):
        return ClickStatus.Valid


class AdRequestAnalyzer(ClickAnalyzer):
    def __init__(self, ruleset='normalRules.txt'):
        super().__init__()
        self.decisiveRules, self.indicativeRules = self.__getAdRequestRules(ruleset)

    def __getAdRequestRules(self, ruleset):
        decisiveRules = []
        indicRules = []

        decisiveRules.append(HumanTimerRule())
        decisiveRules.append(BlacklistRule())
        decisiveRules.append(AcceptLangRule())
```

```
        with open(ruleset, 'r') as f:
            for line in f:
                if line.startswith('DNTRule'):
                    weight = float(line.split('=')[1])
                    indicRules.append(DNTRule(weight))

        return decisiveRules, indicRules

    def analyze(self, request, reqDeque=None):
        reports = []

        isDecisive = True
        weight = 0
        status = ClickStatus.Valid
        for rule in self.decisiveRules:
            name = type(rule).__name__
            if name == 'HumanTimerRule':
                result = rule.testClick(request, reqDeque)
            else:
                result = rule.testClick(request)
            reports.append( RuleReport(name, isDecisive, weight, result) )
            if result == ClickStatus.Fraud:
                status = ClickStatus.Fraud

        sum = 0
        weightSum = 0
        isDecisive = False
        for rule in self.indicativeRules:
            name = type(rule).__name__
            result = rule.testClick(request)
            reports.append( RuleReport(name, isDecisive, rule.weight, result) )
            if rule.weight > 0:
                sum += rule.weight * result.value
                weightSum += rule.weight
            else:
                sum += -rule.weight * result.value

        if weightSum > 0 and sum / weightSum < 0.5:
            status = ClickStatus.Fraud

        return status, reports


class OnlineClickAnalyzer(ClickAnalyzer):

    def __init__(self, ruleset='normalRules.txt'):
        super().__init__()
        self.decisiveRules, self.indicativeRules = self.__getOnlineRules(ruleset)

    def __str__(self):
        string = ''
        for rule in self.rules:
            string = ''.join([ str( rule ), '\n' ])
        return string

    def __getOnlineRules(self, ruleset):
        decisiveRules = []
```

```
        indicRules = []
        with open(ruleset, 'r') as f:
            for line in f:
                if line.startswith('UserAgentRule'):
                    weight = float(line.split('=')[1])
                    indicRules.append(UserAgentRule(weight))
                elif line.startswith('JavascriptEnabledRule'):
                    weight = float(line.split('=')[1])
                    indicRules.append(JavascriptEnabledRule(weight))
                elif line.startswith('RedirectTimeRule'):
                    weight = float(line.split('=')[1])
                    indicRules.append(RedirectTimeRule(weight))
        return decisiveRules, indicRules

    def analyze(self, request, prevDate=None):
        reports = []

        isDecisive = True
        weight = 0
        status = ClickStatus.Valid
        for rule in self.decisiveRules:
            name = type(rule).__name__
            result = rule.testClick(request)
            reports.append(RuleReport(name, isDecisive, weight, result))
            if result == ClickStatus.Fraud:
                status = ClickStatus.Fraud

        sum = 0
        weightSum = 0
        isDecisive = False
        for rule in self.indicativeRules:
            name = type(rule).__name__
            if name == 'RedirectTimeRule':
                result = rule.testClick(request, prevDate=prevDate)
            else:
                result = rule.testClick(request)
            reports.append(RuleReport(name, isDecisive, rule.weight, result))
            sum += rule.weight * result.value
            weightSum += rule.weight

        if weightSum > 0 and sum / weightSum < 0.5:
            status = ClickStatus.Fraud

        return status, reports
```

OfflineDefense.py

```
import time

from onlineDefense import *
from Persistence import *

class OfflineHandler(threading.Thread):
```

```
def run(self, loop=True):
    self.loop = loop
    self.handle()
    pass

def handle(self, ruleset='normalRules.txt', latestFile='latestVariables.txt',
           maxReqs=0):
    self.loop = True
    while self.loop:
        self.prnt('Offline Analysis started')
        behaviorWeight, timePeriodWeight = self.__initWeights(ruleset=ruleset)

        # Load the latest analysis dates
        prevAnalysis = self.__loadLatestFile(latestFile=latestFile)

        # Start off Rule objects and the Database Logger
        behaviorRule = BehaviorRule(weight=behaviorWeight)
        timePeriodRule = TimePeriodRule(weight=timePeriodWeight)
        pagesLoadedRule = PagesLoadedRule()
        dbLogger = DatabaseLogger("dbname=postgres host=localhost port=5432 "
                                  "user=postgres password=postgres")

        # Get the newest requests
        query = 'select * from requests where req_date > ' + \
                str(prevAnalysis) + ' and path = \'/adRequest.html\';'
        newRequests = dbLogger.genericFetch(query)
        reqDB = None
        count = 0
        for newReq in newRequests:
            count += 1
            if maxReqs > 0 and maxReqs <= count:
                break
            reqDB = RequestFromDB(newReq)
            self.prnt2(str(reqDB))
            statusPagesLoaded = pagesLoadedRule.testClick(reqDB, newRequests)
            statusBehavior = behaviorRule.testClick(reqDB)
            statusTimePeriod = timePeriodRule.testClick(reqDB)

            repId = reqDB.report_id
            if repId is not None and repId != 'null':
                reportQuery = 'select * from rules where report_id=' + \
                              str(repId) + ';'
                dbReports = dbLogger.genericFetch(reportQuery)[0]
                reports = self.makeReports(dbReports, statusPagesLoaded,
                                           statusBehavior, statusTimePeriod,
                                           behaviorWeight, timePeriodWeight)
                dbLogger.updateRules(reports=reports,
                                     reportId=''.join(['\'', str(repId), '\'']))

                # Change the current requests' status, if needed
                sum = 0
                weightSum = 0
                newStatus = ClickStatus.Valid
                for report in reports:
                    if report.isDecisive:
                        if report.result == ClickStatus.Fraud:
                            newStatus = ClickStatus.Fraud
```

```
                    else: # Indicative Rule
                        if report.weight > 0:
                            sum += report.weight * report.result
                            weightSum += report.weight
                        else:
                            sum += -1 * report.weight * report.result

                if weightSum > 0 and sum / weightSum < 0.5:
                    print(bcolors.debug2 + str(sum) + '/' + str(weightSum) +
                          ' = ' + str(sum / weightSum) + bcolors.end)
                    newStatus = ClickStatus.Fraud

                if newStatus != reqDB.status:
                    # Change status on the DB
                    statusStr = '\'valid\'' if newStatus == ClickStatus.Valid \
                            else '\'fraud\''
                    query = 'update requests set (status) = (' + statusStr + \
                            ')\nwhere report_id = ' + str(reqDB.report_id) + ';'
                    dbLogger.genericInsert(query)

        # Change latest date in the file to be equal to last request's date
        if reqDB is not None:
            self.__setLatestFile(latestFile, reqDB)

        time.sleep(60)
    return


def __setLatestFile(self, file, req, resetDate = '2018-04-05 00:00:01.000000'):
    with open(file, 'w') as f:
        print('--------------------')
        print(req)
        print('   ******** ')
        try:
            print(req.date)
        except Exception as exc:
            print(exc)
        finally:
            print('   ******** ')
            try:
                f.write(''.join(['prevAnalysis=\'', str(req.date), '\'\n']))
            except Exception as exc:
                print(exc)
                f.write(''.join(['prevAnalysis=\'', resetDate, '\'\n']))
            print('--------------------')
    return


def __initWeights(self, ruleset):
    behaviorWeight = 1
    timePeriodWeight = 1
    # Load the necessary weights
    with open(ruleset, 'r') as f:
        for line in f:
            if line.startswith('BehaviorRule'):
                behaviorWeight = float(line.split('=')[1])
            elif line.startswith('TimePeriodRule'):
```

```
                timePeriodWeight = float(line.split('=')[1])
    return behaviorWeight,timePeriodWeight

def __loadLatestFile(self, latestFile):
    # Load the latest analysis dates
    prevAnalysis = None
    with open(latestFile, 'r') as f:
        for line in f:
            if line.startswith('prevAnalysis'):
                prevAnalysis = line.split('=')[1]
    return prevAnalysis

def __countdown(self, num):
    for i in range(num):
        print(str(num-i) + '...')
        time.sleep(1)

def prnt(self, string):
    string = str(string)
    print(bcolors.offline1 + string + bcolors.end)

def prnt2(self, string):
    string = str(string)
    print(bcolors.offline4 + '\t' + string + bcolors.end)

def makeReports(self, dbReps, statusPagesLoaded, statusBehavior,
                statusTimePeriod, weightBehavior, weightTimePeriod):
    reports = []
    names = ['DNTRule', 'JavascriptEnabledRule', 'HumanTimerRule',
             'BlacklistRule', 'UserAgentRule', 'RedirectTimeRule',
             'AcceptLangRule', 'BehaviorRule', 'PagesLoadedRule',
             'TimePeriodRule']
    isDecisive = [False, False, True,
                  True, True, False,
                  True, False, True,
                  False]
    print(' >>> ' + str(len(names)))
    i = 0
    for rep in dbReps:
        print(str(i) + ' : ' + str(rep))
        i += 1
    print(' >>> ' + str(len(names)))
    for i in range(len(names)):
        name = names[i]
        j = (i*2)+1
        weight = dbReps[j]
        self.prnt2(name + ' &&& ' + str(dbReps[j+1]))
        print(type(dbReps[j+1]))
        if dbReps[j+1] == 'True' or dbReps[j+1] == 'true' or dbReps[j+1] == True:
            result = ClickStatus.Valid
        else:
            result = ClickStatus.Fraud
        if name == 'PagesLoadedRule':
            report = RuleReport(name=names[i], isDecisive=isDecisive[i],
                                weight=0, result=statusPagesLoaded)
        elif name == 'BehaviorRule' and statusBehavior is not None:
            report = RuleReport(name=names[i], isDecisive=isDecisive[i],
```

```
                                weight=weightBehavior, result=statusBehavior)
            elif name == 'TimePeriodRule':
                report = RuleReport(name=names[i], isDecisive=isDecisive[i],
                                weight=weightTimePeriod, result=statusTimePeriod)
            else:
                if weight is None or result is None:
                    continue
                else:
                    report = RuleReport(name=names[i], isDecisive=isDecisive[i],
                                weight=weight, result=result)
            reports.append(report)

        return reports
```

Persistence.py

```
import psycopg2
import threading
from Request import ClickStatus, bcolors

class Logger():
    def __init__(self, name = ''):
        self.name = name

    def log(self):
        return True


class DatabaseLogger(Logger):
    __shared_state = {}
    def __init__(self, connectStr, name=''):
        super().__init__(name)
        self.__dict__ = self.__shared_state
        self.dbLock = threading.RLock()
        self.conn = None
        self.connectStr = connectStr
        try:
            self.conn = psycopg2.connect(connectStr)
        except Exception as exc:
            raise exc


    def logRequest(self, req, status):
        self.dbLock.acquire()
        try:
            self.conn = psycopg2.connect(self.connectStr)
            cur = self.conn.cursor()

            insertStr = """
                insert into requests(ip_address, path, req_date, referer,
                user_agent, accept, accept_encode, accept_lang, dnt, cookies, status)
                values """
            dnt = 'NULL' if req.dnt < 0 else "\'" + str(req.dnt) + "\'"
            valuesStr = ', '.join(["(\'" + req.ip + "\'", "\'" + req.path + "\'",
```

```
                                    "\'" + str(req.date) + "\'",
                                    "\'" + req.referer + "\'",
                                    "\'" + req.user_agent + "\'",
                                    "\'" + req.accept + "\'",
                                    "\'" + req.accept_encode + "\'",
                                    "\'" + req.accept_lang + "\'", dnt,
                                    "\'" + req.cookiesStr + "\'",
                                    "\'" + convertStatus(status) + "\');"])

            cur.execute(''.join([insertStr, valuesStr]))
            self.conn.commit()
            cur.close()
        except Exception as exc:
            print(exc)
            raise(exc)
        finally:
            try:
                self.dbLock.release()
            except RuntimeError:
                pass
            if self.conn is not None:
                self.conn.close()
                print('Database connection closed.')

    def logClick(self, req, reports, status, prevReportId=None):
        self.dbLock.acquire()
        try:
            try:
                self.conn = psycopg2.connect(self.connectStr)
            except Exception as exc:
                print(bcolors.debug2 + str(exc) + bcolors.end)
            cur = self.conn.cursor()
            if prevReportId is None:
                # Log the rules' report for this request
                queryStr = self.__makeRulesQuery(reports)
                if queryStr != '':
                    cur.execute(queryStr)

                # Fetch the latest rule's report_id
                if queryStr != '':
                    queryStr = 'select max(report_id) from rules;'
                    cur.execute(queryStr)
                    row = cur.fetchone()
                    id = int(row[0])
                    report_id = "\'" + str(id) + "\'"
                else:
                    report_id = prevReportId
                print('\033[1;35;46m' + str(report_id) + '\033[0m')

            else:
                # Change the current rules entry
                report_id = prevReportId
                queryStr = self.makeRulesAlter(reports, report_id)
                cur.execute(queryStr)

            # Finish by logging in the request, which is associated with the latest rule row.
            insertStr = """
```

```
            insert into requests(ip_address, path, req_date, referer, user_agent,
             accept, accept_encode, accept_lang, dnt, cookies, report_id, status)
            values """
        dnt = 'NULL' if req.dnt < 0 else "\'" + str(req.dnt) + "\'"
        if req.path.startswith('/adRequest/h'):
            path = "\'/adRequest.html\'"
        else:
            path = "\'" + req.path + "\'"
        valuesStr = ', '.join(["(\'" + req.ip + "\'", path,
                               "\'" + str(req.date) + "\'",
                               "\'" + req.referer + "\'",
                               "\'" + req.user_agent + "\'",
                               "\'" + req.accept + "\'",
                               "\'" + req.accept_encode + "\'",
                               "\'" + req.accept_lang + "\'", dnt,
                               "\'" + req.cookiesStr + "\'", report_id,
                               "\'" + convertStatus(status) + "\');"])

        cur.execute(''.join([insertStr, valuesStr]))
        self.conn.commit()
        cur.close()
    except Exception as exc:
        print(exc)
        raise(exc)
    finally:
        if self.conn is not None:
            self.conn.close()
            print('Database connection closed.')
        try:
            self.dbLock.release()
        except RuntimeError:
            pass

        return report_id

def __makeRulesQuery(self, reports):
    insertStr = 'insert into rules('
    valuesStr = ') values ('

    if not reports:
        return ''

    for report in reports:
        prefix = self.__getRuleQueryPrefix(report.name)
        insertStr = ''.join([insertStr, prefix, 'weight, ', prefix, 'result, ' ])
        valuesStr = ''.join([valuesStr, str(report.weight),
                             ', ', str(report.result), ', '])
    # Remove comma + space at the end of both strings
    insertStr = insertStr[:-2]
    valuesStr = valuesStr[:-2]
    valuesStr = ''.join([valuesStr, ');'])
    finalStr = ''.join([insertStr, valuesStr])
    print('\033[1;36;45m' + finalStr + '\033[0m')
    return finalStr

def makeRulesAlter(self, reports, prevReportId):
    insertStr = 'update rules set('
```

```
        valuesStr = ') = \n('
        endStr = ''.join([') where report_id = ', prevReportId ,';'])

        if not reports:
            return ''

        for report in reports:
            prefix = self.__getRuleQueryPrefix(report.name)
            insertStr = ''.join([insertStr, prefix, 'weight, ', prefix, 'result, '])
            if report.weight is None:
                weight = 'null'
            else:
                weight = str(report.weight)
            if report.result is None:
                result = 'null'
            else:
                result = str(report.result)
            valuesStr = ''.join([valuesStr, weight, ', ', result, ', '])
        # Add extra fields
        insertStr = ''.join([insertStr, 'indicative_score'])
        valuesStr = ''.join([valuesStr, getIndicativeScore(reports)])
        finalStr = ''.join([insertStr, valuesStr, endStr])
        return finalStr

    def updateRules(self, reports, reportId):
        self.dbLock.acquire()
        try:
            try:
                self.conn = psycopg2.connect(self.connectStr)
            except Exception as exc:
                print(bcolors.debug2 + str(exc) + bcolors.end)
            cur = self.conn.cursor()
            query = self.makeRulesAlter(reports, reportId)
            print(bcolors.bluecyan + 'Query is:\n' + query + bcolors.end)
            cur.execute(query)
            self.conn.commit()
        except Exception as exc:
            raise(exc)
        finally:
            if self.conn is not None:
                self.conn.close()
                print('Database connection closed.')
            self.dbLock.release()

    def __getRuleQueryPrefix(self, name):
        prefix = ''
        if name == 'BehaviorRule':
            prefix = 'behavior_'
        elif name == 'UserAgentRule':
            prefix = 'user_agent_'
        elif name == 'DNTRule':
            prefix = 'dnt_'
        elif name == 'TimePeriodRule':
            prefix = 'time_period_'
        elif name == 'PagesLoadedRule':
            prefix = 'pages_loaded_'
        elif name == 'JavascriptEnabledRule':
```

```
            prefix = 'javascript_'
        elif name == 'HumanTimerRule':
            prefix = 'human_timer_'
        elif name == 'BlacklistRule':
            prefix = 'blacklist_'
        elif name == 'RedirectTimeRule':
            prefix = 'redirect_time_'
        elif name == 'AcceptLangRule':
            prefix = 'accept_lang_'
        return prefix

    def genericFetch(self, query):
        self.dbLock.acquire()
        rows = None
        try:
            self.conn = psycopg2.connect(self.connectStr)
            cur = self.conn.cursor()
            cur.execute(query)
            rows = cur.fetchall()
            self.conn.commit()
            cur.close()
        except Exception as exc:
            print(exc)
            raise exc
        finally:
            self.dbLock.release()
            if self.conn is not None:
                self.conn.close()
                print('Database connection closed.')
            return rows

    def genericInsert(self, query):
        self.dbLock.acquire()
        try:
            self.conn = psycopg2.connect(self.connectStr)
            cur = self.conn.cursor()
            cur.execute(query)
            self.conn.commit()
            cur.close()
        except Exception as exc:
            raise(exc)
        finally:
            self.dbLock.release()
            if self.conn is not None:
                self.conn.close()
                print('Database connection closed.')

    def fetchRequests(self, timespan, ip=None ):
        self.dbLock.acquire()
        try:
            self.conn = psycopg2.connect(self.connectStr)
            cur = self.conn.cursor()
            self.conn.commit()
            cur.close()
        except Exception as exc:
            print(exc)
            raise exc
```

```
        finally:
            self.dbLock.release()
            if self.conn is not None:
                self.conn.close()
                print('Database connection closed.')

    def fetchBlacklist(self):
        blacklist = []
        self.dbLock.acquire()
        try:
            self.conn = psycopg2.connect(self.connectStr)
            cur = self.conn.cursor()
            queryStr = 'select ip_address from blacklist;'
            cur.execute(queryStr)
            rows = cur.fetchall()
            for row in rows:
                blacklist.append(row[1])
            self.conn.commit()
            cur.close()
        except Exception as exc:
            print(exc)
            raise exc
        finally:
            self.dbLock.release()
            if self.conn is not None:
                self.conn.close()
                print('Database connection closed.')
            return blacklist

    def fetchBehavior(self):
        return ''
        blacklist = []
        self.dbLock.acquire()
        try:
            self.conn = psycopg2.connect(self.connectStr)
            cur = self.conn.cursor()
            queryStr = 'select ip_address from blacklist;'
            cur.execute(queryStr)
            rows = cur.fetchall()
            for row in rows:
                blacklist.append(row[1])
            self.conn.commit()
            cur.close()
        except Exception as exc:
            print(exc)
            raise exc
        finally:
            self.dbLock.release()
            if self.conn is not None:
                self.conn.close()
                print('Database connection closed.')
            return blacklist

    def __getRequestAttributes(self, req, status):
        return (req.ip, req.path, req.referer, req.user_agent,
                req.accept, req.accept_encode, req.accept_lang,
                req.dnt, status)
```

```
def getIndicativeScore(reports):
    sum = 0
    weightSum = 0
    for report in reports:
        if not report.isDecisive:
            if report.weight > 0:
                weightSum += report.weight
            if report.result:
                if report.weight > 0:
                    sum += report.weight
                else: # report.weight < 0
                    sum += -report.weight
    if weightSum > 0:
        return str(sum/weightSum)
    else:
        return str(0)


def convertStatus(status):
    if status == ClickStatus.Valid:
        return 'valid'
    elif status == ClickStatus.Fraud:
        return 'fraud'
    return ''


def clearLog(filename):
    with open(filename, 'w') as f:
        f.write('\n')
```

Apêndice D

Códigos HTML e Javascript da Rede de Anúncios

AdRequest.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="refresh" content="0; url=http://localhost:8881/redirect.html">
    <title>Title</title>
</head>
<body>
    <p>Click registered. Please wait...</p>
    <script>
        document.cookie = "JSEnabled=true; Path=/redirect.html;";

        function show_image(src, width, height, alt) {
            var img = document.createElement("img");
            img.src = src;
            img.width = width;
            img.height = height;
            img.alt = alt;

            document.body.appendChild(img);
        }

        show_image("http://localhost:8881/shouldLoad.png", 20,20, "shouldLoad");
    </script>
</body>
</html>
```

Redirect.html

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
    <meta charset="UTF-8">
    <meta http-equiv="refresh" content="1; url=http://localhost:8882/">
    <title>Title</title>
</head>
<body>
    <p>You're being redirected to the announcer's page...</p>
    <!--
    <p style="display:none"><img hidden src="http://localhost:8881/hidden.jpg"></p>
    -->
    <div hidden style="display:none;
                       width:0;
                       height:0;
                       visibility:hidden;
                       background:url('http://localhost:8881/hidden.png')">
    </div>

</body>
</html>
```

404.html

```
<HTML>
    <HEAD>
        <TITLE>ERROR: 404</TITLE>
    </HEAD>

    <BODY BGCOLOR="FFFFFF">
        Page doesn't exist!
    </BODY>
</HTML>
```

adDisplayer.js

```
var url = "@AD_LINK@";

document.getElementById("adLink").href = url;
document.getElementById("adImage").src = "http://localhost:8881/announcerAd.png";
document.getElementById("adImage").width = "320";
document.getElementById("adImage").height = "180";
```